Investigation into the Efficacy of Deterministic Minesweeper Solvers

Kostas Kvietinskas
May 2021

Supervisor: Daniel Reidenbach

Abstract

Minesweeper is a game which requires solving a coNP-complete decision problem on each turn to play optimally. The solvability of Minesweeper using deterministic methods (only guessing when necessary) is only partially understood. Furthermore, little is known on the importance of certain information available to the player in solving a Minesweeper board.

In this report we propose a number of hypotheses regarding the capabilities of deterministic solvers for Minesweeper. To investigate these, we present two deterministic solvers: a claimed optimal solver, and a 'naive' solver which attempts to replicate the typical deductive strategy of a diligent human player. These solvers are played on numerous games with differing conditions, with the addition that the optimal solver also has varying informational constraints. Their performances are measured, analysed and compared, and the results discussed with a focus on providing answers regarding the hypotheses made.

It is found that the optimal deterministic solver, even with strict informational constraints (limiting view of the board to areas of 5x5 to 7x7 tiles, and disregarding the total mine count), still manages to win many games. The naive solver, unexpectedly, managed to win nearly as many as the optimal solver even when using a weaker version of its algorithm in which its decision making process is stopped prematurely.

Acknowledgements

I would like to thank Daniel Reidenbach, whose feedback, guidance, and genuine interest in the project has been invaluable.

CONTENTS

1 Background and Literature Review **1**
- 1.1 Introduction . . . 1
  - 1.1.1 Motivation . . . 1
- 1.2 What is Minesweeper . . . 2
- 1.3 Hardness of Minesweeper . . . 3
- 1.4 Deterministic vs Probabilistic Play . . . 3
- 1.5 Single Point Strategy . . . 4
- 1.6 Disjoint Sections . . . 4
- 1.7 Minesweeper as a Constraint Problem . . . 5
- 1.8 Minesweeper with Linear Algebra . . . 7
- 1.9 Other Solvers . . . 7

2 Hypotheses, Methodology and Tools **9**
- 2.1 Hypotheses . . . 9
  - 2.1.1 Information restriction . . . 9
  - 2.1.2 Game Configuration and Agents Abilities . . . 9
- 2.2 Formal Methodology . . . 10
- 2.3 Tools . . . 10

3 A Minesweeper Implementation **12**
- 3.1 Decision to Build a New Implementation . . . 12
- 3.2 Design . . . 13
  - 3.2.1 Components . . . 13
- 3.3 Reproducibility . . . 13
- 3.4 Testing . . . 15

4 Sample Solver Framework **16**
- 4.1 Solve Sample Method . . . 16
- 4.2 Sample Structure . . . 17
- 4.3 Searching the Board . . . 18
- 4.4 Full-Grid Sample . . . 20
- 4.5 Optimisations . . . 20
  - 4.5.1 Skipping Samples . . . 20
  - 4.5.2 Focusing on Frontier Tiles . . . 23

5 Building an Optimal Deterministic Sample Solver **24**
- 5.1 Design . . . 24
- 5.2 Conditions for Correctness . . . 24
- 5.3 Constraint Solver . . . 25
- 5.4 Solving a Sample . . . 25
  - 5.4.1 Mine Count Constraint . . . 26
- 5.5 Optimisations . . . 28
- 5.6 Testing . . . 29

6 Naive Deductive Algorithm **30**
6.1 Exploring the Naive Deductive Process with Constraints . . . 30
6.1.1 Solving Simple Constraints . . . 30
6.1.2 Creating Sub-Constraints to Solve Harder Cases . . . 31
6.1.3 Generalised method for creating sub-constraints . . . 33
6.2 Algorithm idea . . . 33
6.3 Implementation . . . 34
6.3.1 First Implementation . . . 34
6.3.2 Second Implementation . . . 35
6.3.3 Third Implementation . . . 37
6.3.4 Extensions to the Algorithm . . . 38
6.4 Algorithm Pseudocode . . . 39

7 Experiments **42**
7.1 Overview . . . 42
7.2 Running Experiments . . . 43
7.2.1 Tasks . . . 43
7.2.2 Multiprocessing . . . 44
7.2.3 Hardware Specifications . . . 44
7.3 Optimal Solver Experiment Design . . . 45
7.3.1 Parameter Choices . . . 45
7.3.2 Database . . . 50
7.3.3 Handling Interrupts . . . 51

8 Results and Discussion **52**
8.1 Comparison of Solvers . . . 52
8.2 Optimal Solver Results . . . 55
8.2.1 Win Rates . . . 55
8.3 Naive Solver Results . . . 62

9 Conclusion **65**
9.1 Hypotheses . . . 65
9.2 General Remarks . . . 66

Appendices **68**

1 Background and Literature Review

1.1 Introduction

Overview In this project we explore the capabilities of AI that use a deterministic approach to solving Minesweeper. A few hypotheses are made regarding the performance of deterministic solvers under certain conditions and are investigated throughout the project. Two solvers are provided and played on a number of games whilst making various measurements. These are then analysed and the results are discussed.

To be able to play these solvers, an implementation of Minesweeper is developed with support for AI agents.

1.1.1 Motivation

Trying to play a game of Minesweeper optimally requires repeatedly solving a coNP-complete decision problem. Such decision problems are inherently interesting and Minesweeper provides an avenue for investigating them.

The problem with trying to investigate brute-force style algorithms for NP-hard and coNP-hard problems is that it can be infeasible because any algorithm that does so can be expected to have an exponential worst-case time complexity (unless a constructive proof of P = NP exists, in which case we would just opt for an efficient algorithm).

Minesweeper happens to be a small enough of a problem where the current technology widely available is enough to cope with this and allow for experiments running on millions of games within reasonable time-frames.

The aim is to explore the effects of certain design decisions of an AI agent for Minesweeper on its ability to solve it. It is not clear how much of the information available to the player is necessary to solve Minesweeper, or the effects limiting such information would have. As far as is known, there are no available works that look into this to a reasonable degree.

These kinds of insights can inform the design decisions of algorithms made to solve such problems. For example, in the context of Minesweeper, there does not exist (to my best knowledge) any implemented learning algorithm that performs as well as the non-learning solvers out there, especially on the hardest difficulty - Expert mode. Knowing the amount and type of information that is useful could help pave the way to more successful learning algorithms.

Given the hardness of Minesweeper, how effective is a human player's decisions compared to that of AI player capable of finding solutions through a brute-force approach? I.e., just how much of an edge does being able to brute-force search solutions really give over that of what a human could do?

We explore such questions on the differences between a human player and an optimal deterministic player by designing and implementing a 'naive' algorithm (Chapter 6). The naive algorithm attempts to use the style of reasoning that the average Minesweeper player may be expected to use.

1.2 What is Minesweeper

Minesweeper is a computer game played on a rectangular board of tiles that are initially covered. Hidden underneath these tiles are a number of mines. The objective of the game is to uncover all the safe tiles without stepping on any of the hidden mines. If a mine is uncovered, it explodes and the game is lost.

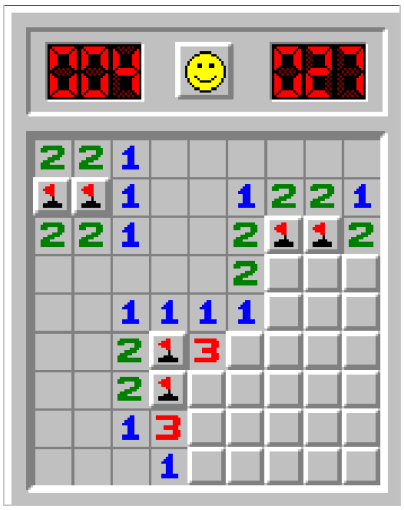


Figure 1.1: Game of Minesweeper on the beginner difficulty

Each open tile gives a clue on the locations of the mines on the board in. Specifically, they give information on the exact number of mines present in the 8 tiles adjacent to it, shown as a number 1-8 the uncovered tile, or left blank if no mines are adjacent. The player is also given the number of mines present in the entire board. Using this information, the player deduces where the mines are hidden, or may be hidden, and then clicks to uncover the tiles they believe to be safe. When all safe tiles have successfully been uncovered, the player wins.

To describe a particular board's difficulty, we use the notation wxhxm to describe a board that is w tiles wide, h tiles high, and contains m mines.

Minesweeper contains three difficulties: Beginner (9x9x10), Intermediate (16x16x40), and Expert (30x16x99). The first click is always safe, and for modern versions this first click is always a 0 (which means all adjacent tiles are safe too).

Note that earlier Minesweeper versions (before 2007 Windows Vista version[13]) have some significant differences, namely, the beginner board is 8x8x10 and the first click is not necessarily a 0 (although it is still always safe). These are relevant due to their impact on a given board's difficulty and need to be considered when trying to make comparisons between results from different sources.

1.3 Hardness of Minesweeper

Verifying if a Minesweeper board is consistent has been proven to be NP-complete[5]. Although it is possible to play Minesweeper by solving the Minesweeper consistency problem, it is not strictly required. What is necessary, however, is solving another decision problem that has been proven to be coNP-complete[11], and so Minesweeper is still hard nonetheless.

This coNP-complete problem asks whether a particular tile on a valid Minesweeper board can be determined to have a definite solution. If it can, then only one of the solutions (it is a mine or it is safe) leads to a valid board, otherwise both solutions are feasible.

1.4 Deterministic vs Probabilistic Play

In Minesweeper, eventually there comes scenarios where you just have to make a guess. Without guessing, you would not be able to continue the game.

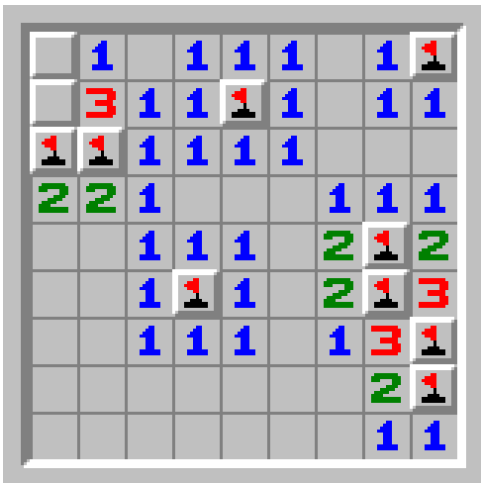


Figure 1.2: A turn of Minesweeper where you have to make a guess

In this report, we differentiate between a deterministic approach to solving Minesweeper from a probabilistic one. A deterministic approach is one that does not guess unless it is absolutely necessary for it to in a given scenario, i.e., when it cannot figure out any moves based on the information available to it. When the deterministic agent guesses, it must do so randomly. An agent that makes an informed (non-random) guess is a probabilistic agent, not a deterministic one.

Note that the idea of when it is absolutely necessary for an agent to guess depends on the decision making processes of the agent, and so it is not necessary for it to make the optimal moves; a solver can still be deterministic if it can't figure out some moves that are in-fact determinable so long as its guesses are random.

1.5 Single Point Strategy

Perhaps the simplest example of a strategy for solving Minesweeper is the Single Point Strategy (SPS). It is a set of rules for deciding whether the adjacent tiles around a single point (an open tile) contain mines or can be opened, but only using the information within that 3x3 area.

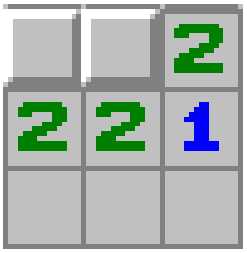


Figure 1.3: A case that can be solved using the single-point strategy

For an open tile with n adjacent mines and k adjacent non-flagged tiles, and f adjacent flagged tiles, we have the SPS rules:

- If $n = k + f$ then all adjacent tiles contain mines and should be flagged
- If $n = 0$ then all adjacent tiles do not contain a mine and are safe to open

The SPS strategy repeatedly applies the above rules to all open tiles until no new moves can be found. Note that flagging and/or opening tiles can impact the decision on other single points which is why this is an iterative process whereby each single point is repeatedly examined until no new moves are found.

The SPS strategy has reasonable success on the easier boards but struggles to win games on the harder difficulties.

1.6 Disjoint Sections

The idea of disjoint sections can be found in other works under various names, such as zone-of-interest[9]. In this project we refer to them as disjoint sections. The disjoint sections contain all the useful information from the grid for solving minesweeper. They can be thought of as the tiles that lie on the edges between open and closed tiles on the board.

Each section is disjoint in the sense that the solutions deduced in one section have no impact on any other disjoint section during that turn. A disjoint section consists of two parts:

- Frontier - closed non-flagged tiles adjacent to an open tile
- Fringe - open tiles that are adjacent to the frontier tiles

The frontier tiles are the tiles we want to solve, and the fringe provides the information of the number of mines that are contained within the different parts of the frontier.

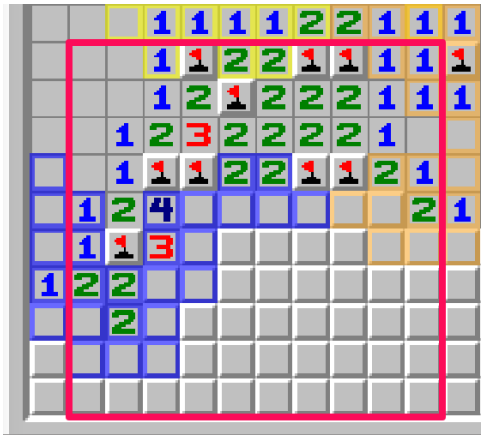


Figure 1.4: Example of 3 unique disjoint sections found from a 10x10 sample

For each open tile with adjacent frontier tiles (closed and non-flagged), we say that that open tile and its adjacent frontier tiles are part of one disjoint section. If a frontier tile is adjacent to multiple fringe tiles, then all of these tiles are part of the same disjoint section.

We make use of samples in this project, i.e., restricted views of the board, and so the disjoint section definition is altered slightly to accommodate for this. In particular, we assume any tile outside the sample that is also adjacent to an open non-zero tile inside the sample to be a frontier tile, and likewise, that open tile is a frontier tile.

This ends up giving us disjoint sections like those seen in Figure 1.4, where some of the tiles to solve (frontier tiles) exist just outside the view of the agent. This is what we want as it is sometimes possible to solve those tiles. Details on how to solve such samples can be found in Chapter 5.

1.7 Minesweeper as a Constraint Problem

A common approach to creating AI agents that solve Minesweeper is to model the game as a constraint problem[1, 2].

We can represent each tile on a Minesweeper board with a boolean variable $x \in \{0,1\}$ which represents whether or not the tile contains a mine or not.

$$x = \begin{cases} 1 & \text{if tile contains a mine} \\ 0 & \text{if tile is safe} \end{cases}$$

Although x is technically not the tile itself (a tile could have extra information such as the number of adjacent mines), we do not enforce such a distinction in the terminology going forward as it is much simpler to just refer to x as the tile itself, and talk about whether the tile x contains a mine or not.

Each open tile displays a number (or is blank if it is a 0) denoting the number of mines

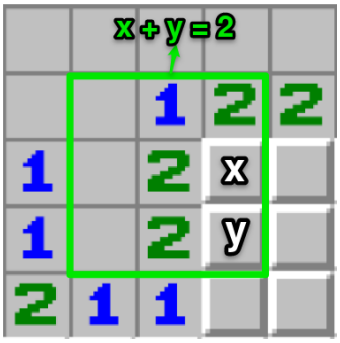


Figure 1.5: Adjacency constraint example

Figure 1.6: Adjacency constraint with flags

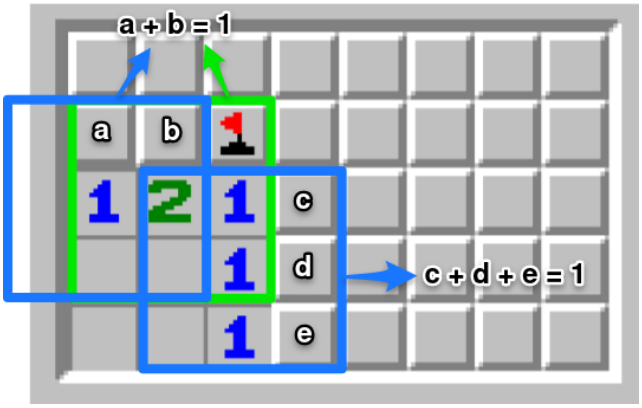


Figure 1.7: Custom-sized board represented by adjacency constraints

that are adjacent to that tile. We can represent this information by building a constraint $x_1 + ... + x_n = y$, where x_i represents one of the adjacent tiles, and y is the number on the open tile. We have that $1 \leq n \leq 8$ as there can be at-most 8 adjacent tiles to an open tile, and if an adjacent tile x_i is open and does not contain a mine (implying $x_i = 0$) it is not necessary to include the variable in the constraint and can be safely omitted. Similarly, tiles that are flagged ($x_i = 1$) can also be omitted from a constraint, but this time we also make sure to decrement the constraint's target sum by 1 for each flagged tile.

By building such adjacency-constraints for each and every open tile with a closed and not-flagged tile, we effectively encapsulate all the information represented by a Minesweeper board and have modelled the problem of solving Minesweeper as a constraint satisfaction problem. One other crucial piece of information that is missing here is the total mine count. That can be represented with a single constraint that constraints every single closed and non-flagged tile on the board (i.e., it constraints all unique variables in the constraints collection built up from adjacency-constraints) and the target sum is the mine count. With the adjacency constraints and the mine count constraint together, solving a turn of Minesweeper is fully modeled as a constraint satisfaction problem.

Note that in this report we use $|V|_{mines}$ to denote the number of mines within a set V boolean tile-variables:

$$|V|_{mines} = \sum_{x \in V} x$$

TYPES OF CONSTRAINTS Not everything on the screen is useful to the player for solving a board. Useful information that can help with solving Minesweeper comes from two places:

1. The tiles on the board
2. Total mine count (counting number of non-flagged mines remaining)

The main thing that we can glean from the board are the numbers on open tiles indicating the number of mines adjacent to them, which can be thought of as a constraint as to the number of mines that must exist adjacent to that open tile.

The mine count can be thought of as a constraint that tells us exactly how many hidden mines must exist in the remaining closed, non-flagged tiles.

Putting the two together, we say that a board can be represented with two types of constraints:

1. Adjacent mines constraint
2. Mine count constraint

1.8 MINESWEEPER WITH LINEAR ALGEBRA

Other solvers exist, such as ones that model Minesweeper with a linear set of equations (constraints) in a matrix and treating it as a linear algebra problem[8, 10]. Gauss-Jordan elimination is then used on the matrix. Often, complete solutions cannot be gained from this (as we often cannot find all solutions to every tile in a single turn), and so a technique is employed to extract partial solutions from the resulting matrix.

Win rates for games which required no guessing are 74.09%, 43.432%, 1.707% for beginner (9x9x10), intermediate (16x16x40), and expert (30x16x99) respectively. This strategy has a modest performance, but performs especially poorly against Expert boards.

1.9 OTHER SOLVERS

There exist a reasonable number of Minesweeper AI implementations. Here we briefly discuss those whose performance results are relevant to the project and have been used (either for comparison or to inform decision made).

BEST SOLVERS Given the hardness of Minesweeper, it is not surprising that many solvers largely rely on heuristics, sacrificing win-rates for faster decision times. The lack of any theoretical investigations into such a limit makes it difficult to determine what the exact win-rates are for an optimal player using the standard Minesweeper boards.

However, there are a couple of solvers[7, 6, 4] which do try to maximise win-rates. Both independently achieved similar win-rates, especially for the games without guessing (within

<1% of each other). Although the overall win-rates did differ somewhat more, on further discussion the authors both managed to work out the details and their solvers began to get win-rates that agreed with each other.

The fact that they independently managed to achieve very similar scores for the 0-guess games, each with a considerably large sample of games (100k and 200k), makes it reasonable to believe that their claimed percentages are the best (or close to the best) win rates achievable playing minesweeper deterministically. We use these win rates as a benchmark throughout the project, assuming them to be the expected win rates of an optimal deterministic solver.

The win percentages are around 92%, 72%, and 16% for beginner (9x9x10), intermediate (16x16x40), and expert (16x30x99) difficulties respectively.

When allowed to guess (not random, good guesses were made to try and maximise win rates), these win rates increased up to 97%, 89%, and 54% for beginner, intermediate, and expert difficulties respectively.

These solvers were run with a fixed first-click position, which has been shown to lead to higher win-rates[6]. The best position, according to one of the solver's experiments[6], varies depending on the difficulty but in all cases it is somewhere near the corner of the board.
The solvers both used the more modern rule first-click rule which ensures that the first click is a 0 tile, meaning all the adjacent tiles get opened up too.

From the solvers that already exist, a cap on the performance of any agent taking a purely deterministic approach (no guesses at all) using all the information available can already be found by looking at the 0-guess win-rates. If the deterministic agent is allowed to

[Insert figure of won game number of guesses distribution data]

RULE-INDUCTION LEARNER There have been attempts to create learners that solve Minesweeper well too, however these still do not perform as well as their non-learning solver counterparts.

Mio learner with best rule set scored 52% on a beginner 8x8x10 board[9] The rule set encompasses rules that are pretty familiar to any player of Minesweeper. It is SPS with the addition of further deducing which tiles are safe based on shared tiles (but it cannot determine if those tiles are mines, not unless it is SPS).

The rules are akin to how a human player would play Minesweeper, and its style of play we term the 'naive deductive approach' which is further explored in Chapter 6. The low win-rate of the learner may suggest that this sort of approach would not perform well, implying that this naive, almost human-like, style of play cannot perform well unless you employ some sort of brute-force tactics.

However, the rule-set used by the learner did not allow it to figure out which tiles are mines for more complicated cases (it could only do so for SPS cases, i.e., in 3x3 areas); it could only figure out that they were safe to click on. It could be that the type of rules used had the potential to perform well but the learner just didn't learn a complete enough set of such rules to really benefit from them.

2 Hypotheses, Methodology and Tools

2.1 Hypotheses

Here are listed several different hypotheses about my beliefs of the capabilities of deterministic solvers with regards to certain factors (e.g., taking away its ability to flag tiles). These serve as the basis for the investigations done throughout the project in an effort to try and provide some empirical evidence in support of or in opposition to these stated hypotheses.

2.1.1 Information restriction

Hypothesis 1 An optimal deterministic agent with largely restricted information can still win nearly as many games as one with all the information available. Only after severe limitations are reached will there be a drastic drop in win-rates.

Hypothesis 2 Restricting the information of the total mine count to the optimal deterministic solver will have a significant impact on the win rate, especially when the solver can see a larger portion of the board.

Hypothesis 3 Restricting information will have the most pronounced effect on the harder difficulties.

2.1.2 Game Configuration and Agents Abilities

Hypothesis 4 Using the first-click-is-always-zero rule will have a small but measurable impact on the win rate of a deterministic solver.

Hypothesis 5 Removing the ability to flag tiles from a deterministic agent will only impact its win rate when its view of the board is restricted. Furthermore, the less of the board the solver can see, the more dramatic the benefit of using flags becomes.

Hypothesis 6 Allowing the deterministic agent to make a random guess will result in an appreciable but meager increase in win-rates from the 0-guess win rates (say, an increase of roughly 4% across all difficulties). Thus the deterministic agent falls considerably short of the capabilities of a well-implemented probabilistic agent.

Hypothesis 7 An algorithm using a naive, human-like deductive strategy will perform poorly in relation to the optimal deterministic solver. Giving this a more measurable objective, I estimate the win rates to be roughly 65%, 30%, 5% for Beginner (9x9x10), Intermediate (16x16x40), and Expert (30x16x99) boards respectively.

2.2 Formal Methodology

Any reasonably sized software development task may likely need some structure over how it is developed. This is where a software development framework comes in handy.

An incremental development approach was taken throughout most of the development of the project. The main parts of the software required were broken down roughly into these separate components, listed in the order that they were worked on:

- Minesweeper implementation
- Visual game renderer
- Non-visual game 'renderer'
- Optimal deterministic agent
- Experiments script
- Naive algorithm agent

Some of these components depend on others (e.g., the renderers require for the Minesweeper game to be implemented first) which dictated the order in which most of these increments were developed. The completion of each individual component, and those before it, can be considered 'complete' in their own regard. Later increments only expand on what has been built before and provide something new, but are not required for the other increments previously finished to function.

However, the formal methodology did however become rather unsuitable in some areas in the context of this project as the challenge of the development of the software was not in the large number of features or changeability of it - there were few requirements and they were pretty rigid and obvious as to what they were - but rather the difficulty was in figuring out how to accomplish a certain task, such as how to build an agent that both plays Minesweeper optimally (deterministically) that can still run efficiently enough to run millions of games so that it can be experimented with.

The use of testing is often a very prominent of such formal methodologies too, but however in this project the use of more rigorous approaches to testing (beyond that of exploratory testing) oddly enough had a negative impact on productivity and were eventually abandoned. What turned out to be more crucial was being able to deal with bugs as they initially came, which was more of an issue of the more lower-level details such as the particular software architectures used and having the appropriate techniques and tooling for debugging.

2.3 Tools

Most significant tools used in the completion of the main content of the project (e.g., software development and running experiments):

- Git and GitHub (code versioning)
- Visual Studio Code (code text editor)
- Python (programming language)
- Jupyter notebook (for data analysis)
- Percona MySQL database
- Google's CP-SAT solver (constraint problem solver that is part of their OR-Tools collection[3]
- Exclusive access to two extra computers (accessed remotely)

One of the two machines mentioned was used for running processor-intensive experiments uninterrupted for long periods of time. The other was used to host a database server to store experiment data.

For project management and report writing:

- Evernote (creating and tracking notes)
- Overleaf (web-based LaTeX editor for report writing)

In particular, there was heavy use of Evernote to keep track of notes for the project, which encompassed a variety of things such as planing, daily journalling, writing to-do lists, documenting decisions made, elaborating on ideas, reviewing literature, keeping track of relevant works, etc.

Adopted the practice of (mostly) daily journalling pretty early which was helpful, not only because it recorded the decisions that were made throughout the project, but also because it helped a lot with day-to-day planning.

Most of the tools used were online tools which allowed for storage on the cloud. This had the benefit of serving as a way to ensure any data related to this project is not lost due to uncertain events which could cause all local files to be lost (e.g., all code was stored in a GitHub repository). Something did end up happening half-way through the project's timeline that resulted in losing all data on my home PC where the majority of project work was done. Thankfully, no work of value was lost as it was all online given the selection of tools used.

There was a good variety in the tools used and they worked well in meeting the various needs of the project. Any more tools would have probably had a negative effect. For example, there was an attempt to use Trello early on to help manage the software development side of the project, however its use to the project did not go beyond what Evernote was already managing to provide.

3 A Minesweeper Implementation

3.1 Decision to Build a New Implementation

Besides Microsoft's various official versions of Minesweeper, there exist various other programs that replicate Minesweeper, either entirely or with a slightly different purpose in mind. There are some versions that are used for competitive Minesweeper, allowing players to record their games which can then be submitted online in competitions to compare with others. There are other Minesweeper implementations geared towards allowing people to build and run their own AI agents to play Minesweeper.

Although these other implementations were considered at the start of the project, many of them either did not seem well implemented or did not quite fit the needs of the project (e.g., lack of ability for agent to flag tiles). Therefore, the choice to build a new implementation of Minesweeper was made.

The obvious advantages of using another implementation instead of building one is that it saves time by not having to spend the initial chunk of time to develop the system. It is common wisdom to not reinvent the wheel. But problems can arise when the wheels available don't quite fit the needs of the project, and are complex enough that attempting to change/fix them would in itself require a significant dedication of time and effort (which could potentially end up being worse than just creating the system from scratch).

The main disadvantages of adopting another system for this project seemed to be:

1. Lack of system understanding could lead to difficulties with modifications (bug fixes, feature extensions, etc.) and understanding behaviours.
2. A lot of the implementations were developed using programming languages that I am not comfortable with.
3. None of the implementations fully met all the needs of the project (e.g., ability for agent to flag tiles)

Point 2 would also worsen point 1, and it would have either forced the use of a language that would have taken required time to learn (and slowed development), or would require time writing a wrapper that allowed for it to be compatible with another language. Neither options were good.

If the use of another implementation was going to include a large overhead in development time and/or slowdown development speed throughout the project, then it just seemed more worthwhile to start from scratch and build a system that will be more familiar to me, allowed for flexibility in design, and full-control over it all.

Overall, it seemed more beneficial to go and implement minesweeper from scratch and use that version for the rest of the project.

3.2 Design

3.2.1 Components

The system is modular. The requirement for AI support using any agent requires an interface that is not implementation-specific. We want to be able to interchange between agents (or to let the user themselves play), to interchange renderers so we can easily switch between

The system is split in such a way that game itself is separate from and is agnostic to how the game is represented (e.g., it could be displayed visually like the regular minesweeper, be played as a console application, etc.). It is the responsibility of the renderer to handle how the game is displayed (if at all) and to handle interactions with the player (agent or human). One responsibility of the executor is to be the interface between the game and any outside components. It takes in an action as input, processes it on the game, and outputs the data about the game in its current state.

The executor is the interaction point for the Game with any outside components, through

A simplistic interface for agents is built by creating an abstract class, from which a user can let their agent class inherit, requiring them to implement a few specific methods, such as `nextMove`, in which the agent should decide and return a move represented by a tuple `(x, y, is_mine)`.

The components of the system interact within the main loop of the program, which is structured roughy as follows:

```
1: while result is not NULL do
2:     renderer.updateFromResult(result)
3:     action = renderer.getNextMove()
4:     result = executor.make_move(action)
```

The executor takes an action and returns a tuple (game grid, mines left, game state). When all games are finished, the executor is depleted and returns NULL whenever it is given any action.

3.3 Reproducibility

Reproducibility is an important aspect of experimentation. An experiment consists of running an agent on a bunch of games using the implemented Minesweeper version. Thus, reproducibility of the experiments rests in the reproducibility of an agent's play within the Minesweeper implementation.

Classical computers are deterministic. The program too would be deterministic (and therefore the results reproducible) if there were no randomness to it. To make the experiment results reproducible, we need to tame this randomness. We do this through the use of seeds and random-generators. Given the same seed, a generator will produce the exact same sequence of outputs, therein lies the mechanism to retain 'randomness' in the program whilst also allowing us to repeat that exact same randomness if we wish.

There are a few places were randomness is used between the Minesweeper implementation and the solver:

- Grid generation - mines are placed down randomly
- Exact sequence of grids (games) that are played in a run of games
- Solver making a guess move by clicking on a random closed non-flagged tile (first click, or when no moves are found in a turn)

Each game is generated through a game seed. The same game seed should lead to the same underlying grid being made. Note that due to the rule of first-click being safe, the same game seed can end up having different underlying grids depending on the first-click's position and whether the rule of first click always having 0 adjacent tiles is being enforced. No one underlying grid is really the 'true' grid of a specific game seed (even if they all end up near-identical), and so in practice a grid is dependant on having the same game seed and the same first click (position and specific first-click rule used).

Run seed determines the sequence of games that are played. Specifically, a particular run seed is used to generate a sequence of n game seeds that are to be used in a run of games, where n is the number of games specified. The run seed is ignored when
Game seed determines the board that is generated at the start of a game.
Run seed determines the run of games that will be played.

A run seed determines the run of games to play by generating N sequential game seeds, where N is the number of games provided by the user. Given the same run seed, the same N game seeds will be generated in the exact same order.
The game seed is used to generate the board at the beginning of the game (on first click, grid is populated with mines based on first click's position and whether the rule for first-click-is-zero is enabled or not). The exact same game seed will generate the exact same board, regardless of run seed.

A given game seed always produces the same grid, regardless of the agency seed, the run seed that was used to generate it, or the sequence of games that were generated before it. If someone wishes to reproduce a certain set of games, they can simply do so by feeding in the list of game seeds at the start that they want to be played and let it run. They can even do the same and switch out the agent that they use to try and compare how the two perform. This also helped with debugging algorithms in the solver as when it crashed, the game seed can be extracted and then the agent can be replayed on that exact same game to investigate what the problem was.

3.4 TESTING

Unit testing framework was established. It however turned out to be a waste of time and was abandoned as the minesweeper implementation hardly changed after the initial development and so did not need any regression testing. Major bugs were caught early on through deliberate exploratory testing. Establishing the test framework and writing up test cases ended up costing development time and provided almost no benefit.

It seems the reason more formal methods of testing had little benefits here is that the requirements of the implementation were rigid, few, and pretty obvious - it had to be a reasonably efficient replica (of the important features) of Minesweeper with basic support for AI agents.

4 Sample Solver Framework

In this project, we wanted to limit an agent's information by narrowing its view of the board into some variably sized sample. For this, there was a need for some sort of framework for getting all useful samples from a board on each turn and feeding them into the sample-solving algorithm(s).

Initially, this was built in conjunction with, and solely for, the optimal solver strategies (Chapter 5) but due to its design being agnostic to the details of the sample-solving strategies used, its subsequent reuse with the naive algorithm (Chapter 6) was straightforward. From this, it became clear that the framework was better thought of conceptually as a separate entity and so the chapters have been split in such a way as to reflect this.

4.1 Solve Sample Method

This method is the heart of the sample solver. Inside it, we place whichever strategies we wish for solving a given sample and return all unique moves acquired.

```
1: function SolveSample(sample)
2:     disjointSections = getDisjointSections(sample)
3:
4:     moves_1 = strategy_1(sample)
5:     moves_2 = strategy_2(disjointSections, self.mine_count)
6:     ...
7:     moves_n = strategy_n(disjointSections, moves_2)
8:
9:     return moves_1 ∪ ... ∪ moves_n
```

A sample is initially scanned for its disjoint sections (see Sub-Chapter 1.6 for details) and fed into the strategies that make use of it. Partitioning the input sample into such disjoint sections can lead to speed-ups for particular algorithms, and maybe even simplify them as they do not need to scan the sample themselves for relevant tiles as the disjoint section already provides this.

As multiple strategies may wish to use these disjoint sections, we avoid having to unnecessarily reprocess the disjoint sections multiple times by doing that in this method instead of putting that responsibility on each algorithm.

Each algorithm may make use of the sample or disjoint sections, and potentially extra information (such as the current mine count).

In this particular implementation, the framework is part of a class and all the extra information are fields of the class which we can access in python with the `self.` prefix.

To promote decoupling, we deliberately pass all the information that a strategy requires

through its function parameters at this point. That way each algorithm is not reliant on the class and has no side effects, i.e., the algorithm neither affects nor depends on the state of the program, it only returns a specific answer when given specific inputs. This way, an algorithm is easier to fix if errors do occur; from experience, state-dependant methods with potential side effects are often significantly harder to debug than those without.

4.2 Sample Structure

A sample is a rectangular portion of the board. Along with the usual tile properties, a sample may also include wall tiles (denoted by python's `None` type in the framework implementation) so that it can capture the useful information that a sample is on the border of the board.

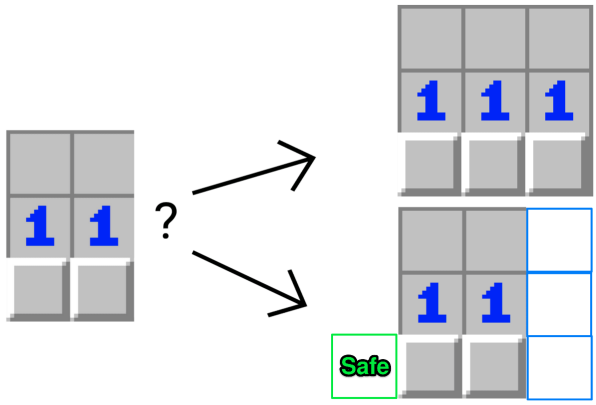


Figure 4.1: Illustration of how wall tiles in a sample can lead to a solution

The inclusion of the knowledge that tiles are adjacent to a wall can allow for some moves to be determined. This can be seen in Figure 4.1, where the move can only be found given the fact that the wall is there.

The relevant information in some area of the board is not necessarily rectangular in shape, and so it seems feasible that an agent may choose to look at non-rectangular areas of the grid when trying to play. However, even these shapes can fit into some minimal rectangular area and would be missed if the view of the agent were limited to a smaller rectangle.

If an agent performs well on an NxM grid but poorly on an $(N-1)$xM grid, then that is a strong suggestion that the dimension size N is about the cut-off point for a significant amount of crucial information for that particular agent (i.e., the particular set of algorithms used). Put another way, this gives us reason to believe that the a lot of the required information from the tiles on a board will be in some shape enclosed within an NxM area (for potentially varying

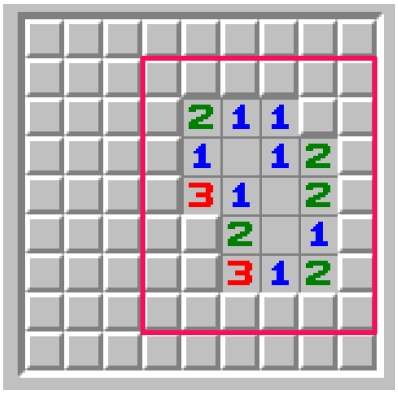


Figure 4.2: Complex-shaped disjoint section captured within a rectangle

M).

The use of rectangular samples allows for easy adjustment of the limit of information (from tiles on the board) placed on an agent whilst also making it simple to reason about and compare the different performances of an agent under varying limits of information.

4.3 Searching the Board

To search for moves on a turn, we iterate over all possible samples of the specified size, starting from the top-left and scanning left-to-right, top-to-bottom.

For each useful sample, we try and solve it using the `SolveSample` method described earlier. If the sample yields one or more new solutions we stop the search and play these moves one after the other.

After the first click has been made, the order in which moves are played does not matter for deterministic play. That means these moves can be played in any particular order without the fear of somehow preventing some solutions from being determinable later in the game.

This makes sense when we consider that playing a move only ever increases the amount of information that can be found in any sample on the board (either through flags or opening up tiles). If a sample can determine a certain tile, adding more information to it (by playing other moves first) will not prevent you from being able to determine that same tile for that specific sample.

When scanning the board and getting samples, we make sure to go along the boundaries too to include the samples with wall tiles as knowing certain tiles are next to the board's boundaries can affect the determinability of tiles. A width of 1-tile of wall tiles in a sample is enough information to let the agent know there's a wall there. Thicker walls provide no benefit

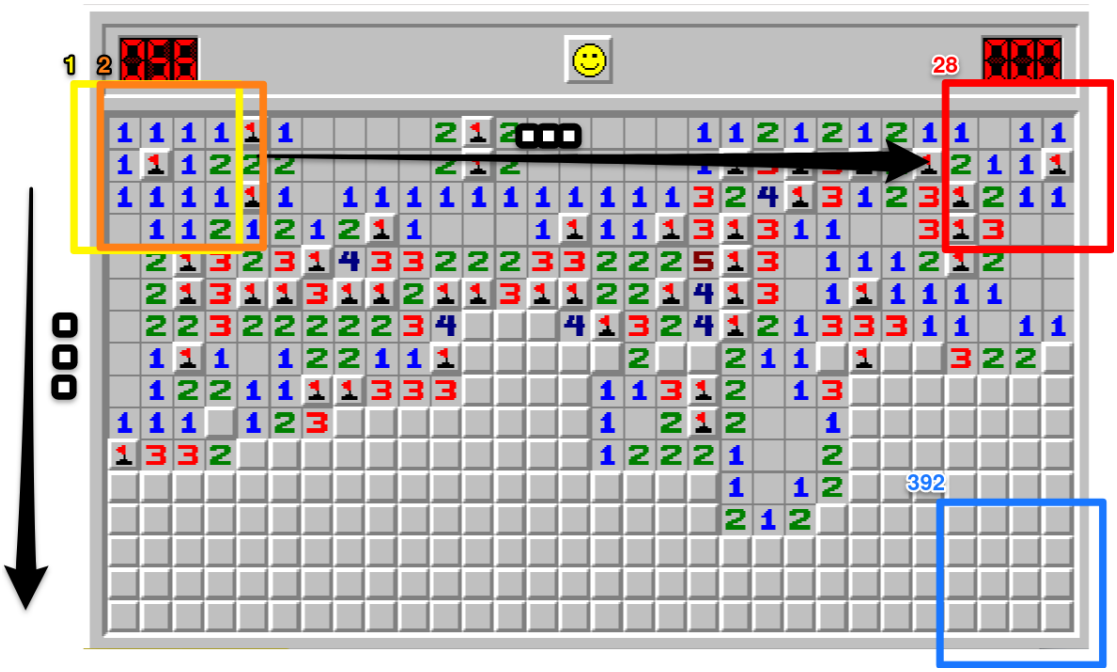


Figure 4.3: Illustration of the order in which samples are iterated over the board

therefore we brush along the edge of the board when iterating to include at-most 1-tile wide wall boundaries in a sample, preventing pointless inflation in the number of samples that we try and solve in a turn.

Figure 4.4: 6x5 sample with 1-tile thick wall

Figure 4.5: 6x5 sample with 2-tile thick wall

4.4 Full-Grid Sample

When playing on grid of size WxH, we would expect all results from sample sizes of WxH or larger to be the same as, at the point, increasing the sample size would not provide the agent with any new information. However, given that we let the solver know about walls by providing a sample with wall tiles, we would in-fact need a sample of size (W+2)x(H+2) to truly represent the full grid.

A more convoluted solution to this would be to give the agent the ability to deduce that a sample is surrounded by wall tiles based on the grid size and sample size. This is not necessary, however, as we can just fix any sample of size WxH to exactly (W+2)x(H+2), thus saving development time whilst also keeping the code complexity low.

More specifically, the implementation supports rectangular sample sizes. The adaption for this requires a minor tweak: when one dimension of the sample is large enough to fit across the entire grid whose size is N in that dimension (e.g., sample is at least as wide as the grid), then that dimension gets fixed to a size of $N+2$.

4.5 Optimisations

Solving a sample is an expensive operation. The main idea underlying the optimisations discussed here is in trying to reduce the number of samples that need to be solved per turn. This is done through either prioritising particular samples first in the hopes of finding solutions earlier in a turn or skipping some samples entirely.

4.5.1 Skipping Samples

Solving every sample on each pass of the board, although simple, can lead to a lot of wasted computations. To cut down on this waste and improve the performance of an agent, we try to minimise the number of samples that are passed into `SolveSample` method. Care needs to be taken into which samples we choose to skip as skipping those that have the potential yield determinable moves will ruin the correctness of the agent, even if the strategies used to solve individual samples are correct.

Two techniques are used here to cut down on the number of samples that are solved in a turn which rely on the following observations:

1. Some samples can never yield determinable solutions
2. A sample that has been solved once does not need to be solved again on later turns if its contents remain unchanged.

Technique 1 Skip samples that we know cannot be solved. We can examine samples to look for particular characteristics that let us know that the sample just cannot be solved to give determinable moves.

We filter out all those samples that do not contain both a non-flagged covered tiles and an uncovered tile located either inside or just outside the sample (but still adjacent to the sample). These sorts of samples simply do not have any useful information for solving it as either there are no tiles that can be solved by the sample, or there is no information to use to solve those tiles.

TECHNIQUE 2 Skip samples that have already been solved in that game. A certain sample at a specific position on the board can remain unchanged between turns. If that sample with its exact contents has already been solved, then there is no need to try solving that exact same sample in the subsequent turns as it contains no change in information.

To implement this, we get the hash of a (sample, sample_position) tuple for each sample we examine. If this hash has not yet been seen in the current game, we store it, otherwise, we skip the sample.

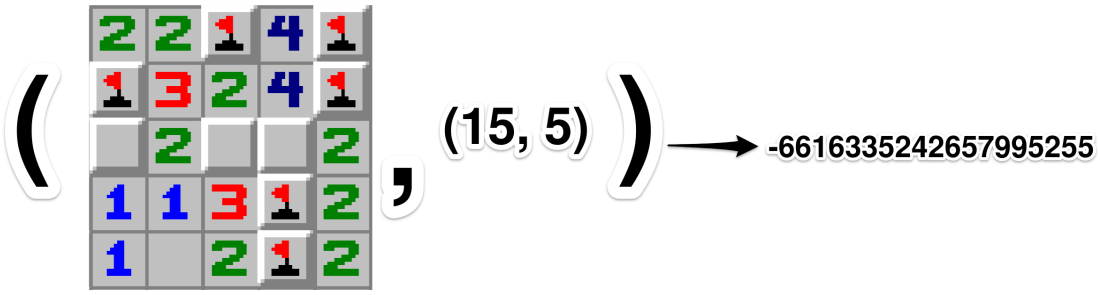


Figure 4.6: A tuple containing a sample and its position being hashed

These hashes are stored in a python `set` data structure which grants the ability to check if a new hash already exists in the collection with an O(1) average time complexity[12].

Without the inclusion of the sample position in the hash, a sample that yields a solution in one area of the board may end up being skipped if it occurs again in another area of the board within the same game, thereby missing moves that can be played.

A workaround to this would be to either keep track of sample-hash & solutions pairs (which can then be reused) or to add an extra check that prevents sample hashes of solution-yielding samples from being added to the collection (thereby preventing that sample from being skipped).

There are an extremely large combination of samples that we can come across, and so the frequency of this happening seemed far too low to warrant the use of these more complete techniques that increase the complexity of the code. To keep the code simpler for readability, we instead opt for including the sample position when hashing a sample.

FAILED ATTEMPT When observing a sample-solving agent play, it became apparent that a lot of time was spent near the beginning of a game iterating over samples around a section with few tiles open. Take for example this section of the board in Figure 4.7

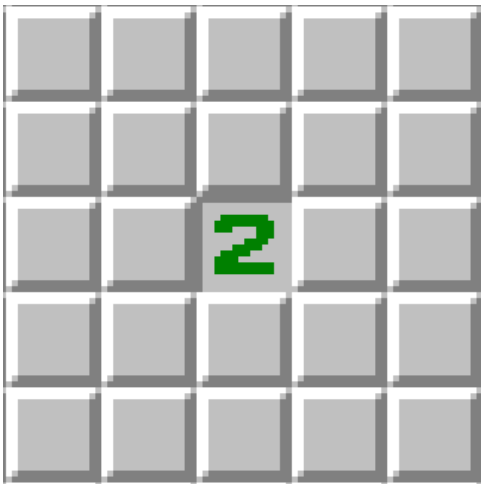


Figure 4.7: A section containing only one open tile

the agent will iterate over every single sample that contains that one open tile. For a 5x5 sample size, that's 25 samples, only 16 of which even contain the entire disjoint section. This is particularly wasteful as only one sample that contains the entire disjoint section is required to solve it. Once that section is solved, we know we can skip all other samples that include that section in its current form.

The technique we used was to keep track of which disjoint sections were fully contained in a solved sample in that game. This is done by hashing the section and storing it in a set, similar to the sample hashes. We then skip any sample that (partially or fully) contains only those sections that have already been solved. We do this as we know the sample contains no new information previously solved samples did not have, and so this sample cannot yield new solutions.

Note that when a disjoint section changes (either new tiles added due to changes in the board or some tiles are flagged), it needs to be solved again. This is why hashing a section works here, as the hash will change when the section changes, and so the changed disjoint section will not be skipped unless a sample containing the entirety of it is solved.

Although it makes sense in theory, on measuring the performance of such an optimisation it was found that the time that was reduced by the optimisation was canceled out by an increase in time required to get the disjoint sections. Therefore, this optimisation seemed to provide no real speed-up to the program and so it was removed to keep the code simpler.

4.5.2 Focusing on Frontier Tiles

A turn ends as soon as the solver finds one or more solutions from a sample (or when all samples have been checked). By changing the order of the samples that are examined in a way that prioritises those more likely to contain determinable moves, we can expect to see a drop in the time it takes the solver to finish a game.

The useful information in a sample is the sections made up of frontier and fringe tiles. It makes sense then that the samples that try to include as much of these areas, i.e., the ones which contain the most of the useable information, are the samples that are more likely to contain a solution. Therefore we prioritise those samples that are roughly centered on the frontiers of the board.

We split a pass over the board into two sequential phases:

1. Iterate over samples that are centered on frontier tiles
2. Iterate over all possible samples

So as to maintain the correctness of the agent, we use the final phase as a catch-all method to ensure no useful samples are accidentally skipped. With the optimisation that skips already-examined samples, we can be sure that the second phase does not attempt to solve those samples already solved in the first phase.

5 Building an Optimal Deterministic Sample Solver

This section discusses a sequence of strategies that is used within the sample solver framework (Chapter 4) to solve a given sample deterministically and, it is believed, optimally. In other words, if a move can be figured out with certainty from a sample using logical reasoning, then this set of strategies should be able to do just that.

5.1 Design

Since an optimal (or at least near-optimal) solver is the goal, we go for the idea of using a brute-force strategy as the final, catch-all-solutions strategy for solving a sample. We can potentially use other efficient strategies first to catch any easy, early solutions before eventually falling back on the brute-force algorithm to catch the remaining ones.

The final design uses the Single-Point Strategy (SPS) followed by the brute-force approach.

```
1: function SolveSample(sample)
2:     disjointSections = getDisjointSections(sample)
3:
4:     spsMoves = singlePointStrategy(sample)
5:     bruteMoves = bruteForceStrategy(sample, disjointSections, spsMoves, mines_left)
6:
7:     return spsMoves ∪ bruteMoves
```

The brute-force algorithm makes use of disjoint sections (see Sub-Chapter 1.6) as an optimisation. This is further discussed later in 5.5 Optimisations.

The SPS moves are provided to the brute-force algorithm to speed it up by cutting down the search space, preventing the need for it to figure out what was already known from SPS.

5.2 Conditions for Correctness

Using constraints to model the board has the nice benefit that it is simple enough in its representation that reasoning about such an optimal solver's correctness gets reduced down to a manageable set of criteria it has to meet, i.e., there is a limited number of places where things could possibly go wrong. This was one of the reasons that a constraints approach was used for the optimal deterministic solver.

By this design, we propose the following four main conditions for correctness of the solver:

1. all samples that could potentially give solutions in a turn are considered
2. the two types of constraints listed are indeed all that is required to play Minesweeper optimally deterministically
3. these constraints are built correctly given any valid sample

4. the constraint solver correctly solves the constraint problem provided

These conditions were not meant as a rigorous and exhaustive list to check against for proving correctness, but rather as an outline of the main areas that could cause the solver to be less than optimal. Thus, its use was found in supporting debugging efforts by making clearer the areas to investigate for errors, and in acting as a guideline to assist in decision-making regarding the design and development of the solver.

For example, consider the decision of whether it was worth switching to using an existing constraint solver or not. An external constraints solver built by experts would be far more likely to satisfy condition 4 than a constraint solver built from scratch within this project. This reason makes this decision far more attractive given the goal of creating an optimal deterministic solver.

5.3 Constraint Solver

Knowing the hardness of solving Minesweeper (see 1.3 Hardness of Minesweeper, it is expected that an algorithm that solves these constraints would be exponential in the worst case. Efficiency is clearly a key concern here as we want to play many games with the solver to experiment on it.

Building a constraint solver from scratch is not the wisest of moves for this project when there already exists other implementations freely available. The solver that we use is Google's CP-SAT solver[3]. It is called CP-SAT because it converts linear constraints into a SAT problem and then runs an efficient SAT solver on it.

Techniques for efficient SAT solvers have many years of research to back it, and the CP-SAT tool has been built by a team of experts. It seems far more likely that such a constraint solver will outperform and be more reliable than anything that could ever be hoped to be produced by a sole developer within the time frame of this project.

Considering all of this, using an existing constraint solver seems an obvious choice. That is why I am ashamed to admit that in the beginning effort was misplaced in attempting to create a brute-force algorithm of my own to solve the very same problem. This lack of foresight cost 2-3 weeks of effort, and was one of the more costly of decisions made throughout the project. Needless to say, the switch to using the CP-SAT solver had resulted in a tremendous leap in progress for the project.

5.4 Solving a Sample

Minesweeper can be modeled as a constraint satisfaction problem, as discussed in Sub-Chapter 1.7. The solver proposed in this section makes use of this idea. We represent a provided sample with a collection of constraints. These are then passed to the constraint solver, from which we look for all solutions that can be found. To build an optimal deterministic solver, we focus on correctly creating these constraints for any sample.

Creating the adjacency constraints is a simple process. Each fringe tile (open-tile number

with adjacent tiles yet to be determined) yields an adjacency constraint. We represent all the adjacent-mines information in a sample by building up collections of all these constraints from all the visible fringe tiles from all the disjoint sections of the sample.

The mine count constraint requires more explanation as it is not so straightforward when using samples like they are when considering the entire grid (in which case we would simply say the sum of all variables from the board equals the total mine count).

5.4.1 Mine Count Constraint

The total mine count tells us how many mines remain hidden in the non-flagged closed tiles on the board. With a sample, we might only see a subset of these tiles, but using information available to the player we can still reason about the range of mines that can/must exist within certain tiles.

Definitions To help with explaining the details of the constraint, we use the term border to refer to those tiles that are part of the frontier but are outside the sample. A frontier tile is adjacent to a visible open tile, all of which are inside the sample, so we have that all tiles in the border are just outside the perimeter of the sample. Note that being outside but adjacent to the sample does not necessarily mean it is part of the border - it has to be a frontier tile.

We split the number of hidden mines m left on the board into three summands: $U+V+W=m$. Specifically:

- U is the number of hidden mines that exist in the sample
- V is the number of hidden mines that exist in the border
- W is the number of hidden mines that exist outside the sample and border

As we cannot see the border tiles, we cannot know whether they are already flagged or not. That means if we decide a tile on the border should be flagged, it may not actually contribute towards the sum V because if it's already flagged, then m will have already been decremented between the turns and we would instead be double-counting that mine. To handle this uncertainty, we split V into two:

$$V = V_F + V_{AF}$$

V_F is the number of mines we propose exist within the frontier tiles on the border based on the information we have, and $V_{AF} \leq V_F$ is the number of those tiles-to-flag that have already been flagged on the actual board.

BOUNDS DERIVATION The tiles which we are trying to constrain are those that contribute towards U and V_F, as they are the ones that exist as part of the adjacency constraints. To help find solutions to these we want to constrain these as tightly as possible with the mine constraint, i.e., we want to find the minimum and maximum values of $U + V_F$.

$$\begin{aligned} U + V + W &= m \\ U + (V_F - V_{AF}) + W &= m \\ U + V_F &= m - W + V_{AF} \end{aligned}$$

Separating out the terms allows us to find the lower and upper bounds of $U + V_F$ by finding the minimum and maximum values of $m - W + V_{AF}$.
We have that m is a constant, and only W and V_{AF} can vary (dependant on the surrounding board the sample was found).
To minimise $U + V_F$ we set W to W_{max}, and set V_{AF} to 0 (every border tile flagged really does count towards m).

$$0 \leq m - W_{max} + 0 \leq U + V_F$$

We assume W_{max} is the exact number of tiles that exist outside the border and sample as we do not know anything more about these tiles and cannot deduce anything about how many of them are hidden and non-flagged.

For the upper bound we assume $W = 0$ and $V_{AF} = V_F$, i.e., all hidden mines exist inside the sample and border and any of the tiles we want to flag just outside the sample are already flagged (and so flagging them does not contribute to m).

$$U + V_F \leq m - 0 + V_F$$

THE CONSTRAINT Let $m_{min} = max(m - W_{max}, 0)$. Putting this and the bounds together we get the constraint:

$$m_{min} \leq U + V_F \leq m + V_F$$

We simplify the constraint slightly to remove the double occurrence of V_F:

$$m_{min} - V_F \leq U \leq m$$

FULL GRID SAMPLES As a sanity check, consider the case when the sample contains the entire grid. We have $W_{max} = 0$ and $V_F = 0$ as all tiles are inside the sample. We calculate

$$m_{min} = max(m - 0, 0) = m$$

Now the mines constraint becomes

$$m \leq U \leq m$$
$$U = m$$

We can see that it has reduced to the full-grid global mines constraint as would be expected.

5.5 Optimisations

Disjoint sections Disjoint sections allow for a strategy to solve each, smaller, disjoint section one at a time as opposed to trying to solve the whole bunch in one go which can lead to shorter run times overall if an algorithm has a non-linear time complexity.

The brute-force strategy makes use of this to reduce the search space for its constraint solver, but only when the agent is ignoring the mine-count. Once the mine count is included, a constraint over all frontier tiles within a sample is required and so it is simpler to just feed all the constraints in a sample in one bunch in that case.

Probing In the constraint solver, we employ a technique known as probing within the Mixed Integer Linear Programming (MILP) community. This was used to significantly speed up the performance of the CP-SAT solver to solve a sample's constraints. To probe boolean variables, we set some variable x to either true or false and investigate its consequences.

In the context of this particular solver, we initially use the CP-SAT solver to search for the first satisfying assignment over the variables for the given constraints. We then use these truth values in the assignment as 'potential definite values' for the variables. For each variable x, we take its potential definite value $y \in \mathbb{B}$ and probe it by setting x to the opposite value, $x = \bar{y}$ and then continue running the CP-SAT solver. There are two possible scenarios from here:

- Another satisfying assignment is found $\implies$ x is not determinable (it can be either a mine or not a mine)
- No satisfying assignment found $\implies$ $x = y$ as $x = \bar{y}$ has been shown to be impossible

In the latter case, we add a new constraint, $x = y$ to the CP-SAT solver's model so that the search space is reduced before moving on to probing the next variable. Note that in the former case, not only does finding a second satisfying assignment prove that x is non-determinable, it can also show that other variables are non-determinable too if their truth value in the second assignment differs from the one in the first assignment, allowing us to skip probing those non-determinable variables when we get to them.

Failed Optimisation - Linear Algebra and Gauss-Jordan elimination A failed attempt at an optimisation was to use another preemptive strategy (after SPS but before brute-force) that models Minesweeper as a linear algebra problem, which is described in Sub-Chapter 1.8. The implementation of the algorithm for use in this solver ended up being far too slow for its inclusion to have any benefit.

5.6 TESTING

Test cases could be literal cases of games with the appropriate solutions. However representing such cases is cumbersome manually unless some tooling is developed to allow for these cases to be drawn up (a sort of level-editor for Minesweeper). This seemed like it require more time to develop than the testing would help save, and so the idea of testing the agent with test-cases was not done.

What was needed were different techniques that can still allow us to tell if the solver is putting out incorrect moves which it thinks are correct, and to tell when it is missing solutions. Instead of creating cases, it was easier to just play the agent on the game itself and then investigate any cases where it fails.

For checking if the agent decides on erroneous moves, a simple assert statement is in place that checks that a game hasn't been lost after playing a move it was sure about.

For checking if the agent missed solutions, this was more difficult as we can never really be sure certain scenarios could even be solved. A simple approach was to manually check whether the solutions the solver came up with matched with what was believed to be correct. This helped in the earlier stages when certain solutions are obvious, but it does not help so much for the less obvious solutions.

The real test for this is by using the other two known solvers 0-guess win rates as the benchmark for this solver. If the completed solver, run on thousands of games, managed to get similar win rates then we have evidence that the algorithm is indeed optimal (for deterministic plays), assuming that the other two solvers are both optimal too for deterministic plays and their 0-guess win rates are indeed the expected win rates for such a solver. This does not, however, confirm that the implementation is correct for when we are restricting information (smaller sample sizes and/or ignoring mine count) as we have nothing to compare those win-rates to. Another problem with this test is that, even if it can indicate a flaw in the implementation and/or algorithm, it provides no real clue as to what that flaw is or how to find it.

The use of the verification test did help with the discovery of two bugs in the solver. Fixing these finally had the solver getting win rates (for 0-guess games) as high as expected.

6 Naive Deductive Algorithm

In this chapter we discuss a natural strategy to playing Minesweeper and how it can be modeled using constraints. Afterwards, an algorithm that encapsulates this natural strategy is provided, generalising what is a pretty straight-forward and intuitive deductive process.

The algorithm is used within the sample solver framework (Chapter 4). Its full pseudocode description is preceded with commentary over some of the key optimisations made along the way that led to the final design.

The naive algorithm was intended to resemble the strategy that would be used by a person playing Minesweeper. It is believed by both the author, and the project's supervisor, that this naive deductive process implemented by the algorithm reasonably models the reasoning a diligent human player would use.

6.1 Exploring the Naive Deductive Process with Constraints

As discussed in Background (Sub-Chapter 1.7, the state of the board can be represented by building constraints of the form $x_0, ..., x_n = y$, where $x_i \in \mathbb{B}$ and $y \in \mathbb{N}_0$. This turns the problem of solving Minesweeper to a problem of finding (partial or whole) solutions that satisfy these constraints.

6.1.1 Solving Simple Constraints

To explain the 'naive' deductive process, let's first begin by going through some simple cases and walking through how to solve them:

This here can be solved using the Single-Point Strategy (SPS) (see Sub-Chapter 1.5) When trying to solve an individual constraint $x_0 + ... + x_n = y$, we can apply the rules of SPS to solve it. Let $V = \{x_0, ..., x_n\}$. SPS rules applied to the constraint are as follows:

- $y = |V| \implies$ all $x_i \in V$ are mines
- $y = 0 \implies$ all $x_i \in V$ are safe

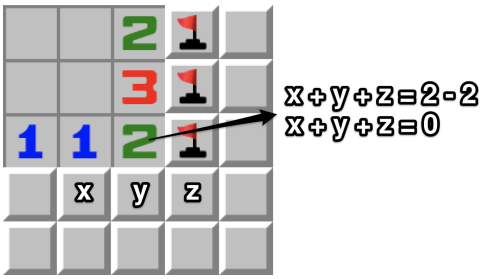


Figure 6.1: source: `https://minesweeper.online/help/patterns`

This simple way of solving constraints (by looking at each individual constraint separately) and applying SPS rules is the constraint solving process that is used in the intuitive deduction process. What is left is the process of figuring out new constraints beyond the ones that can be directly gleaned from the board.

6.1.2 Creating Sub-Constraints to Solve Harder Cases

Consider now these two cases:

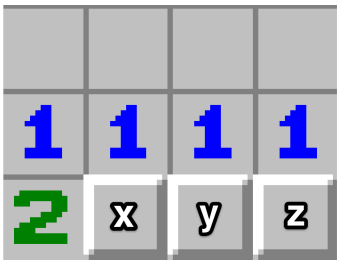


Figure 6.2: 1-1

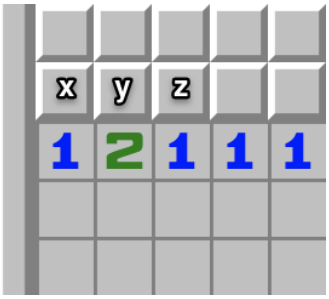


Figure 6.3: 1-2 pattern, source: `https://minesweeper.online/help/patterns`

For these cases, solving individual constraints (applying SPS) is not enough, but clearly these can be solved. The trick here is to consider the tiles that both constraints share and consider:

- how many mines must be inside the shared tiles?
- how many mines must be inside the tiles that are not shared (tiles adjacent to one open tile, but not the other)?

In Figure 6.3 we can create the two constraints:

$$x, y|1$$
$$x, y, z|2$$

We know that they both share x and y which contains exactly 1 mine. We can use this fact to figure out that the rest of the variables in the second constraint (just z) must therefore contain 2 - 1 = 1 mine.

$$\begin{aligned} n &= |x, y, z|_{mines} - |x, y|_{mines} \\ &= 2 - 1 \\ &= 1 \end{aligned}$$

Similarly, for the 1-1 case, the constraints

$$x, y|1$$
$$x, y, z|1$$

allow us to find the new sub-constraint

$$z|0$$

which we can solve to get the solution $z = 0$.

We extend the notation of constraints to accommodate for these ranges of mines: $x_0, ..., x_n|(l, u) := l \leq x_0 + ... + x_n \leq u$ where $l, u \in \mathbb{N}_0$ are the lower and upper bounds, respectively, of the constraint representing the minimum and maximum amount of mines that are contained within the tiles the variables represent.

Progressing onto more difficult cases, sometimes it is necessary to repeat the process of comparing in-between sections (finding sub-constraints) multiple times to get a particular solution.

This idea of repeating the process of comparing certain sections of variables to establish the number of mines there has to be in a new section is the main tool a human player uses to solve a Minesweeper board. An experienced Minesweeper player learns to recognise certain patterns on the board, allowing them to discover solutions from memory, but underlying all these non-brute-force patterns is this same deductive process that can be used to derive such solutions from each of the common patterns.

Note that this intuitive strategy cannot solve all scenarios where a deterministic play is applicable. In these cases you would need to take a more brute-force approach and try out all possible configurations of mines to determine which tiles are definitely safe and which are not. It is these sorts of cases that give you a feel for why Minesweeper is coNP-Hard. Sometimes there just isn't a straightforward process, such as the naive deductive process, that will help you derive the solutions even if those solutions exist.

6.1.3 Generalised method for creating sub-constraints

Here we generalise the process of comparing two constraints to create new sub-constraints. When comparing two constraints $A|l_A, u_A$ and $B|l_B, u_B$, we define

$$S = A \cap B$$
$$\bar{A} = S \cap A$$
$$\bar{B} = S \cap B$$

and we consider 3 new possible sub-constraints based around the tiles that they share:

$$S|l_S, u_s$$
$$\bar{A}|l_{\bar{A}}, u_{\bar{A}}$$
$$\bar{B}|l_{\bar{B}}, u_{\bar{B}}$$

The upper and lower bounds for the constraint over S can be found by first finding those bounds when examining each of A and B:

$$l_{S_A} = max(0, l_A - |\bar{A}|) \qquad u_{S_A} = min(|S|, u_A)$$
$$l_{S_B} = max(0, l_B - |\bar{B}|) \qquad u_{S_B} = min(|S|, u_B)$$

We then reconcile these two mine ranges $[l_{S_A}, u_{S_A}]$ and $[l_{S_B}, u_{S_B}]$, tightening the range if possible:

$$l_S = max(l_{S_A}, l_{S_B}) \qquad u_S = min(u_{S_A}, u_{S_B})$$

Using the mine range $[l_S, u_S]$ over S, we can now figure out the mine ranges of the other two sub-constraints:

$$l_{\bar{A}} = max(0, l_a - u_s) \qquad u_{\bar{A}} = min(|\bar{A}|, u_a - l_s)$$
$$l_{\bar{B}} = max(0, l_b - u_s) \qquad u_{\bar{B}} = min(|\bar{B}|, u_b - l_s)$$

6.2 Algorithm idea

The algorithm repeats the processes of solving individual constraints SPS-style, and creating new sub-constraints by comparing two constraints until all solutions that can be gained from this approach are exhausted. From a simplistic high-level view, it goes as follows:

1: Create new sub-constraints by comparing pairs of constraints and add them to the collection of constraints
2: Try to solve each individual constraint
3: Repeat 1 and 2 until no more solutions can be found.

Note that the algorithm terminates because there are only so many sub-constraints you can create from a finite collection of finite constraints, which means there will eventually be no more new constraints to try and solve to yield such solutions.
We can see that there is a finite number of unique subconstraints that can be created by noticing that each time we compare two constraints, we create at-most 3 sub-constraints each of which constrains at least 1 less variable than the constraint from which its variables originates from.

The stage of solving constraints should take into account any solutions found throughout the process. E.g., if we discover in the process that $x_i = 1$ and $x_j = 0$, then we should include that knowledge when trying to solve other constraints too.

6.3 Implementation

So as to shorten development time, initial implementation was set on making the simplest working version of the algorithm first, then improving it wherever it needs it but only if it needs it (optimisations and bug fixes). This meant reusing code where possible and writing simple but perhaps inefficient code before attempting to complicate things with optimisations.

6.3.1 First Implementation

Having already implemented the other solver, there was already a good framework for the solver at hand with methods such as getting disjoint sections from the board. To implement this algorithm, we reuse the framework of that solver and replace its use of SPS & brute-force strategies with the naive deductive algorithm. Note that, as with the other solver, it is better to work on solving constraints from disjoint sections rather than piling them all together as that reduces the input size of the algorithm. Disjoint sections are those whose constraints are not coupled to any other constraint from a different disjoint section, and so solutions to constraints from one disjoint section have no impact on those from another.

Another bit of code that was reused is a method for building constraints, built earlier for use with another algorithm that solves a constraints matrix via Gauss-Jordan elimination (Sub-Chapter 1.8). Each constraint is represented in the form $[x_1, ..., x_n, y]$ where n is the number of overall constrained variables across all constraints for a given section of the board, $x_i \in \{0, 1\}$ which is 1 if the i'th variable is included in this constraint, 0 otherwise. y is the target sum, i.e, the number of mines that are in the tiles that are included in this constraint (the ones with a 1 in their respective column). E.g., when building constraints from a section with 4 variables v_1, v_2, v_3, v_4, the constraint $[0, 1, 0, 1, 2]$ would represent $v2 + v4 = 2$. Note that at this

point in the implementation, the need for ranges of mines for this algorithm was not realised which is why there is only an exact target in these constraints rather than lower & upper bounds.

For the algorithm control flow, it was split into two main sequential stages: create all sub-constraints first, then solve all those constraints after.

```
1: function NAIVEDEDUCTIVESOLVE(C)
2:     C = FindAndAppendAllSubconstraints(C)
3:     moves = solveConstraints(C)
4:     return moves
```

Getting the sub-constraints simply consisted of repeatedly comparing all possible pairs of constraints and adding the any new constraints to the list of all constraints until this yielded no more new constraints.

```
1: function FINDANDAPPENDALLSUBCONSTRAINTS(C)
2:     do
3:         for each pair of constraints in C do
4:             create sub-constraints from pair
5:             filter out sub-constraints that already exist in C
6:             add remaining new constraints to C, if any.
7:     while new constraints have been added
8:
9:     return C
```

This first implementation turned out to be, unsurprisingly, far too inefficient to use. There are many constraints that can be made from just a small collection of initial constraints, and so this approach of blindly comparing all pairs of constraints repeatedly quickly becomes too cumbersome.

6.3.2 SECOND IMPLEMENTATION

Optimisation key notes:

1. A constraint only needs to contain the variables it constrains and the mine count range
2. Only coupled constraints - those that share variables - need to be compared
3. Some constraints can be solved early on

ITEM 1 We adopt a slimmer representation of a constraint C as a tuple (V, T), where V is the set of variables constrained by C, and T is the range of mines in those tiles referenced by variables in V. We identify each variable x_i by a number i, and so $V = \{i \mid x_i \text{ is constrained by C}\}$. Note that now, for each constraint, $|V| \leq 8$ (usually in the 1-4 region), whereas before a single constraint could, and frequently would, reference dozens of variables. This compaction both simplifies constraint comparisons and speeds it up too.

ITEM 2 Sub-constraints are built by looking at the shared variables between two constraints and comparing mine ranges. If there are no shared variables, then we can't build any sub-constraints from them, hence why we should only try to compare coupled constraints.
It would be cumbersome to search for which constraints are coupled to a certain constraint by naively comparing it with all constraints, so instead we build a mapping R(var) = { constraint | var in constraint }, where var is a constraint variable.
we can now figure out relatively quickly which constraints are coupled with a given constraint like so:

```
1: coupled = ∅
2: for var in constraint do
3:     coupled= coupled ∪R(var)
```

We build this mapping as the constraints are initially being built, and are then passed in as an input to the algorithm alongside the constraints to the naive algorithm.

As the collection of constraints is changed (whether by adding new constraints or by updating new constraints based on discovered solutions), the mapping needs to be updated too to reflect this.

In the actual implementation, the mapping was tweaked to contain constraint references (we just use indexes as references as constraints are in stored in an ordered list) as opposed to the constraints themselves, i.e., R(var) = { constraint_index | var in constraint }. This is done to prevent duplicating constraints across the constraints collection and the mapping, which would have complicated and slowed down the process of updating constraints on new solutions or when new constraints are added. Although it is a meaningful optimisation, it does make the algorithm harder to read and so is omitted in any pseudocode representations of the algorithm in this section as it is not a vital detail.

ITEM 3 The idea is that by finding solutions early, we can shrink constraints down thereby reducing the total number of sub-constraints that can then be made from then on. Less constraints should lead to quicker run times.

To achieve this, we change the flow of the algorithm from the sequential type to an iterative one, exhaustively cycling between solving constraints and finding new sub-constraints until no new sub-constraints can be found (and thus no new moves).

Following the optimisations, the method for creating new sub-constraints is substantially changed. It now only compares coupled constraints, and no longer exhaustively creates all sub-constraints but instead undergoes one iteration when called.

This led to significant performance gains but on particular (frequent enough) inputs it would run far too slow.

```
1:  function NaiveDeductiveSolve(C)
2:      moves = ∅
3:
4:      do
5:          moves = moves ∪ SolveConstraints(C)
6:          C = UpdateConstraintsFromSolutions(C, moves)
7:          C = FindAndAppendNewSubConstraints(C)
8:      while C has new constraints
9:
10:     return moves
```

```
1:  function FindAndAppendNewSubConstraints(C)
2:      for c1 ∈ C do
3:          coupledC = {constraint ∈ C | constraint is coupled with c1}
4:
5:          for c2 ∈ coupledC do
6:              newC = CreateSubConstraints(c1, c2)
7:
8:              for x ∈ newC do
9:                  if x ∉ C then
10:                     C.append(x)                    ▷ Only add new constraints
11:     return C
```

6.3.3 Third Implementation

Although the second implementation did cut back on the number of unnecessary constraints comparisons by only comparing coupled constraints, it still had the flaw that it was needlessly re-comparing constraint pairs that had already been compared.

Optimisation key note:

- Constraint pairs that have already been compared do not need to be re-compared as this will yield no new sub-constraints

A caveat to this is that constraints that have been updated do in-fact need to be treated as if they are new constraints and re-compared with other constraints, otherwise some solutions that this algorithm should figure out can end up being missed.

The solution here was inspired by the idea of the dirty-bit which is used in OS page replacement algorithms. We say that each constraint is either dirty or not dirty. If a constraint is dirty, it is to be compared against all other constraints, after which it becomes no longer dirty. If a constraint is not dirty (has already been examined), then it will be ignored (can still be indirectly examined by being paired with another dirty constraint).

A constraint is set as dirty when either:

- it is a new constraint that has just been added to the constraint collection
- it has just been updated from a solution

Note that from the start, all constraints are considered to be new constraints and so they are all dirty.

```
1:  function FindAndAppendNewSubConstraints(C)
2:      for c1 ∈ C do
3:          if c1 is not dirty then
4:              continue                          ▷ ignore it and continue to next constraint
5:
6:          set c1 as not dirty
7:
8:          for c2 in C do
9:              newC = CreateSubConstraints(c1, c2)
10:
11:             for x ∈ newC do                              ▷ Only add new constraints
12:                 if x ∉ C then
13:                     C.append(x)
```

This led to a significant speedup. This implementation's performance was satisfactory and could be used in an experiment to measure the algorithm's efficacy.

6.3.4 Extensions to the Algorithm

Further optimisation

One flaw in the algorithm that impacts efficiency is that we very frequently need to check whether a constraint already exists in the constraints collection. Since the actual implementation uses a list, this takes O(n) time on each lookup. It would be better if this could be done in O(1) time on average.

With that in mind, one further optimisation would be to use a dict (or some hashmap datastructure) where the variables of a constraint are the key, as opposed to a list of constraints. This should lead to faster run times, however the current implementation is already fast enough and this change would not particularly simplify the algorithm anymore and so, within the context of this project, it was not considered a worthwhile endeavour.

If the reader intends to implement this algorithm and performance is more of a concern, then this is a low-hanging fruit of an optimisation that is likely to yield significant gains and is probably worth trying out.

n-STEP ALGORITHM

The algorithm's iterative process allows for a natural extension that allows for a step-limit input to be provided, which limits the number of iterations that can be made in the main loop, as opposed to iterating to exhaustion (when no more new sub-constraints could be found). This would allow for the comparison between the performances of n-step versions of the algorithm with varying $n \in \mathbb{N}_0$.

Note that for the 0-step version, it should be equivalent to SPS as only the initial constraints are solved (and solving is done in the same manner as with SPS).

6.4 ALGORITHM PSEUDOCODE

```
1:  function NaiveDeductiveSolve(C, stepLimit)
2:      if stepLimit is NULL then
3:          stepLimit = ∞
4:
5:      moves = ∅
6:      step = 0
7:
8:      do
9:          moves = moves ∪ SolveConstraints(C)
10:         C = FindAndAppendNewSubConstraints(C)
11:         step + = 1
12:     while C has new constraints and step < stepLimit
13:
14:     return moves
```

```
1:  function SolveConstraints(C)
2:      allMoves = ∅
3:
4:      do
5:          moves = SolveConstraintsOneIteration(C)
6:          newMoves = allMoves \ moves              ▷ Filter out duplicate moves
7:
8:          if newMoves then
9:              C = updateConstraints(C, newMoves)
10:     while newMoves
11:
12:     return allMoves
```

```
1:  function SolveConstraintsOneIteration(C)
2:      allMoves = ∅
3:
4:      for (V, (l, u)) ∈ C do
5:          if l ≠ u then          ▷ Can't get any moves if we're not sure on exact mine count
6:              continue
7:
8:          numMines = u
9:
10:         if numMines == |V| then
11:             isMine = True
12:         else if numMines == 0 then
13:             isMine = False
14:         else
15:             continue                                   ▷ Can't solve constraint, move on
16:
17:         for x ∈ V do
18:             allMoves = allMoves ∪ {(x, isMine)}
19:
20:     return allMoves
```

```
1:  function UpdateConstraints(C, moves)
2:      for (x, isMine) ∈ moves do
3:          for each c1 ∈ C that contains x do
4:              c1 = GetUpdatedConstraint(c1, (x, isMine))
5:              set c1 as dirty
6:      return C
```

```
function GetUpdatedConstraint(c1, move)
    (V, (l, u)) = c1                                   ▷ unpack constraint
    (x, isMine) = move                                 ▷ unpack move

    V = V \ {variable}
    if isMine then
        l = max(l - 1, 0)                              ▷ Can't be below 0
        u = u - 1
    else
        u = min(u, |V|)

    c1 = (V, (l, u))                                   ▷ pack constraint with updated values
    return c1
```

```
function FindAndAppendNewSubConstraints(C)
    for c1 ∈ C do
        if c1 is not dirty then
            continue                                   ▷ ignore it and continue to next constraint

        set c1 as not dirty
        coupledC = {c2 ∈ C | c2 is coupled with c1}

        for c2 ∈ coupledC do
            newC = CreateSubConstraints(c1, c2)

            for x ∈ newC do
                if x ∉ C then                          ▷ Only add new constraints
                    set x as dirty
                    C.append(x)
    return C
```

Note that the method CreateSubConstraints implements the one generalised method described earlier in the chapter

7 Experiments

7.1 Overview

Experiment 1 - Optimal Solver Experiment

Main experiment for the optimal deterministic solver. Run of 15,000 games per combination of experiment parameters (how much? games total). Detailed data is to be stored in a database, from which a variety of analysis can be done.

Experiment variable parameters, along with their range of values:

- Sample size: 4x4, 5x5, ... , 10x10, Whole-grid
- Use mine count: True, False
- Can flag: True, False
- Difficulty: Beginner 8x8x10, Beginner 9x9x10, Intermediate 16x16x40, Expert 30x16x99
- First click always zero: True, False

Experiment constant parameters:

- Number of games: 15,000
- Agent seed: 4040
- Run seed: 40
- First click position: Random

For clarification, a whole-grid sample means a sample that is large enough to contain the entire grid and the surrounding wall tiles, the exact size of which depends on the size of the grid.

All sample-sizes that are large enough to fit the particular difficulty's grid are only run once as the whole-grid. For example, we do not run the beginner 8x8x10 grid once for each of the sample sizes of 8x8, 9x9, and 10x10. Instead it is run only once. This is done as these differences in sample sizes would have the same results (they have no differences in the information they provide) and would only serve to waste precious time for running the experiment.

Note that the actual number of games was initially planned to be 25,000 and was later reduced to 15,000 per parameter combinations as the experiment was taking too long to be able to complete before the project's deadline.

EXPERIMENT 2 - NAIVE ALGORITHM EXPERIMENT

Experiment variable parameters, along with their range of values:

- Step limit: 0, 1, ..., 4, Unlimited
- Can flag: True, False
- Difficulty: Beginner 8x8x10, Beginner 9x9x10, Intermediate 16x16x40, Expert 30x16x99
- First click always zero: True, False

Experiment constant parameters:

- Number of games: 25,000
- Agent seed: 50
- Run seed: 5050
- Sample size: Whole-grid
- First click position: Random

The step limit describes which n-step version of the algorithm is used. Further details can be found in Chapter ref bit about this.

The naive algorithm is implemented within the sample solver framework (Chapter 4), but only a view of the entire grid is used in this experiment. This is mainly due to the limiting time constraints of the project.

7.2 RUNNING EXPERIMENTS

An experiment script (experiment.py) was made specifically to define and run these sorts of experiments. It makes specifying and running the experiments easy by providing a somewhat consistent interface which had certain helpful features. One of these features allowed an experiment to be defined by providing the range of possible values of parameters (the experiment variables) and to specify which parameters are constant (which are in-turn saved to a separate file to record these, e.g., agent seed).

7.2.1 TASKS

Each experiment is broken down in to a number of tasks, where each task is essentially a batch of games run with an agent with specific parameters for that agent and run of games. As these experiment were likely to run for a long time, a progress bar was used to indicate the proportion of tasks that were complete.

Each run of games, for a specific combination of parameters, is broken down into a series of independant tasks. Instead of completing all the tasks from one run consecutively for a certain combination in one go, we instead play tasks in a round-robin fashion across all the experiment parameter combinations, interleaving them.

Tasks were ordered in this way for two main reasons:

1. It allows for all the variety of games to be played early on, meaning any early results gathered are of those across all experiment parameter combinations (and these just get more precise as the number of games are played increases over time)

2. It provides a stable progress bar completion rate allowing for more accurate completion time estimations.

Point 1 was particularly important for the optimal-solver experiment, where a single run of games could take a long time to complete, and it could be possible that the experiment may have to end early, in which case it was important to get results across all parameter of combinations early, allowing for early analysis to be done, instead of skipping certain parameters combos entirely.

To elaborate on point 2, a specific run of games could take longer to run than another (e.g., expert games vs beginner games). Playing all games from a run consecutively can cause the progress to increase at different rates throughout the experiment, making estimating the time remaining difficult. By using a round-robin selection of tasks across runs, we avoid this problem as these tasks of different run-times are now interleaved.

7.2.2 Multiprocessing

By splitting experiments into a sequence of independant tasks to be executed, applying multiprocessing became a natural step which significantly sped up the run of experiments. This takes advantage of the multi-core processor available on the PC that was used to run the experiments.

This was a particularly rewarding decision that did not require much work to get going as the experiment script was deliberately built with this in mind, hence the splitting of experiments into independant tasks.

Without the use of multiprocessing, the optimal solver experiment would have been impossible to run within the project's timeline in its current shape. Its range of parameter values would have had to be drastically cut, missing a lot of potentially interesting conclusions that could be drawn from the resulting data-set.

7.2.3 Hardware Specifications

Two extra computers were available throughout the project, one of which was used for running these experiments uninterrupted, and the other to host the database server used to store

Component	Description height
OS	Windows 10
CPU	Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.5GHz
CPU Cores	6 physical, 12 logical
RAM	16GB
GPU	GeForce GT 710

Table 7.1: Hardware specifications of the computer used to run the experiments

the results of the optimal-solver experiment. The performance of the solvers used in the experiments are dependant on the exact hardware that these experiment were run on, and so below is listed the hardware specification of the computer that they were run on:

7.3 Optimal Solver Experiment Design

7.3.1 Parameter Choices

First click position First click position was implemented as an option into the solver to help verify win rates of the optimal solver. However, it was decided that it should be left out for the optimal solver experiment. The impact of the first-click position on win rate is known and has been investigated in various other literature[1, 6]. The inclusion of a fixed first click position would increase the number of experiment parameter combinations twice-fold or more. Such a large increase in experiment size was not worth the data it would provide and so was left as random instead.

Number of games per parameter combination The results from the verification test done on the optimal-solver were used to find how quickly the win-rates converge across the difficulties. From Figure 7.1 we can see that the win-rates do converge to reasonable very remarkably quick. Past the first 100-500 games, we see it stabilise to within

By excluding the first 100 games from Figure 7.1, we get Figure 7.2 which shows the convergence for the rest of games more clearly. From this it was concluded that all the win-rates seem to stabilise almost entirely by around 20,000 games. It was therefore planned for the optimal solver experiment to run for 25,000 games (raised up, just to be on the safe side). However, it later became apparent during the run that the experiment would not finish before the project's deadline, so the number of games was reduced to the final value of 15,000 games instead.

Batch size and number of processes An investigation on the optimal number of processes and batch size was done by running a small-scale version of the optimal-solver experiment on the same PC that it was supposed to be run on. Each combination of num_processes and batch_size were run on the exact same set of games, and so the differences in performance times are due solely to the difference in these parameter values.

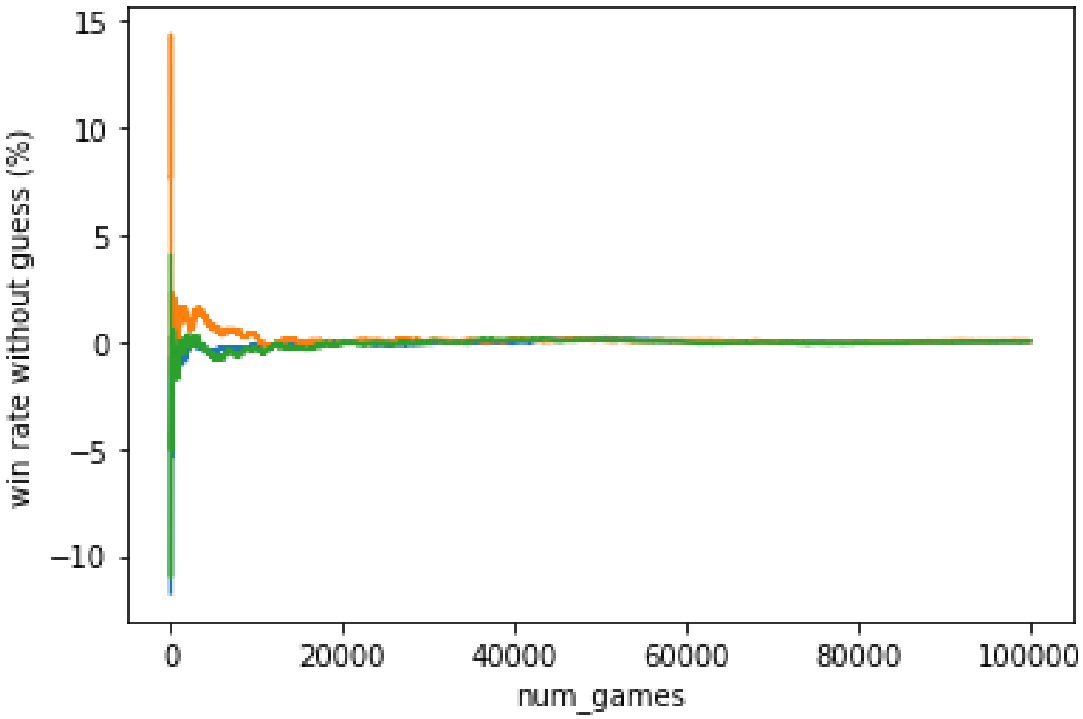


Figure 7.1: Convergence of win-rates from the optimal-solver verification test, all games

Figure 7.3 shows the running times when the results were being saved into a local CSV file. This was done as development was being done on the database during the time this small-scale experiment ran. The intention was to get results over a wide range of values for both parameters, and then refine later when the connection between the optimal-solver experiment and database had been established.

We see that there is a clear increase in speed as the number of processes approaches around 12. The PC has 12 logical cores (8 physical), so it seems that this result coincides with that. Beyond 12, the times seem to vary somewhat randomly. Additionally, it seems that the smaller the batch size, the better

From Figure 7.3, we conclude that the optimal number of processes and batch size here seems to be 12 and 5 respectively. The small batch size was an unexpected outcome as switching between tasks in multiprocessing may be seen as an expensive task. However, it seems that running a sequence of games is slower than starting a completely new run by switching between tasks. This suggests that the switch-over between games within a run is inefficiently implemented.

A possible reason that the optimal batch size is no longer 1 when the database is used may be due to the high frequency of database calls such a small batch size creates, causing processes to have to wait to store the data that they have gathered from executing their task.

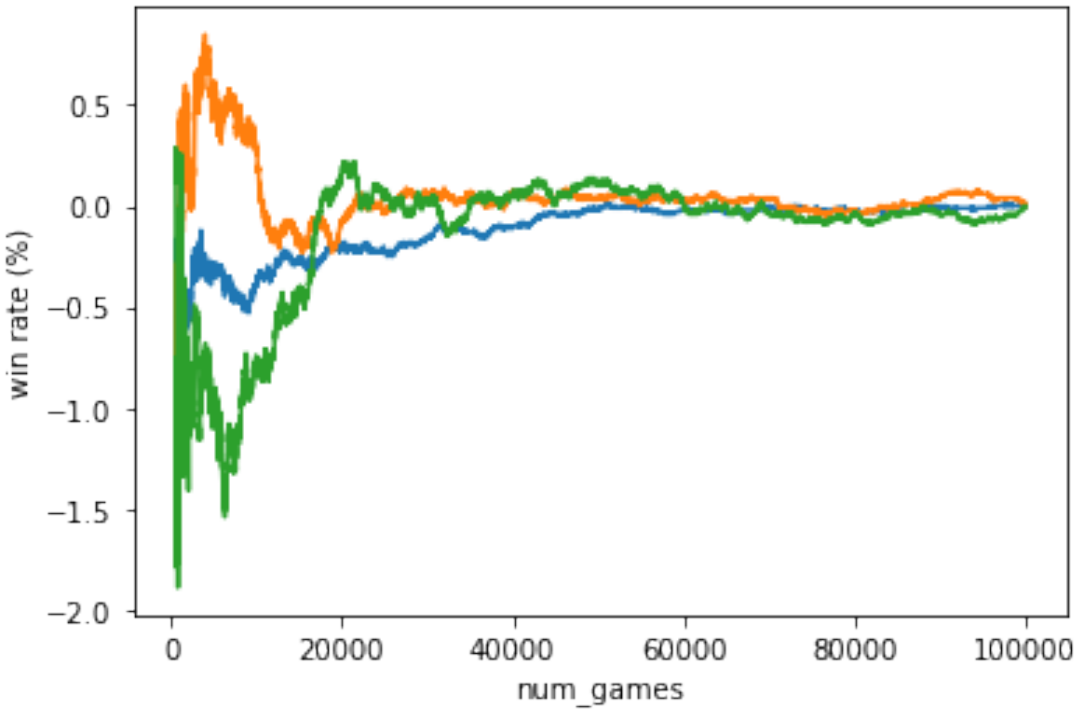


Figure 7.2: Convergence of win-rates from the optimal-solver verification test, excluding first 100 games

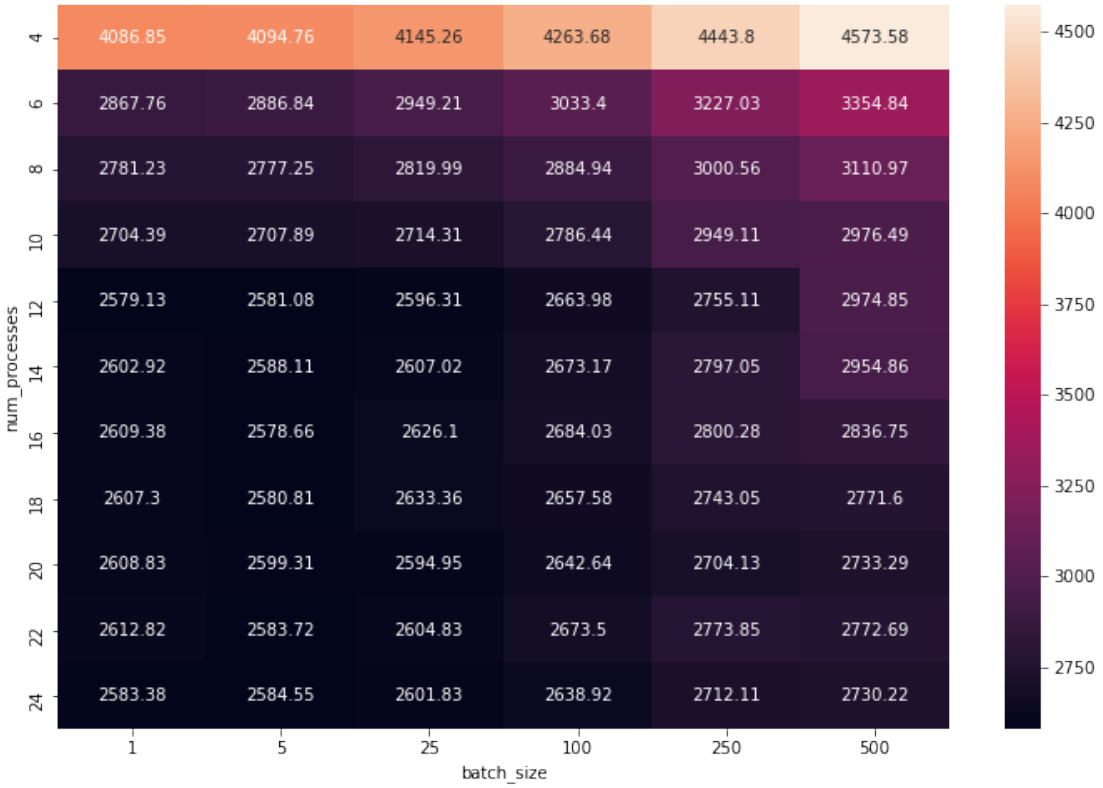


Figure 7.3: Batch v processes for small-scale optimal-solver experiment with local saving

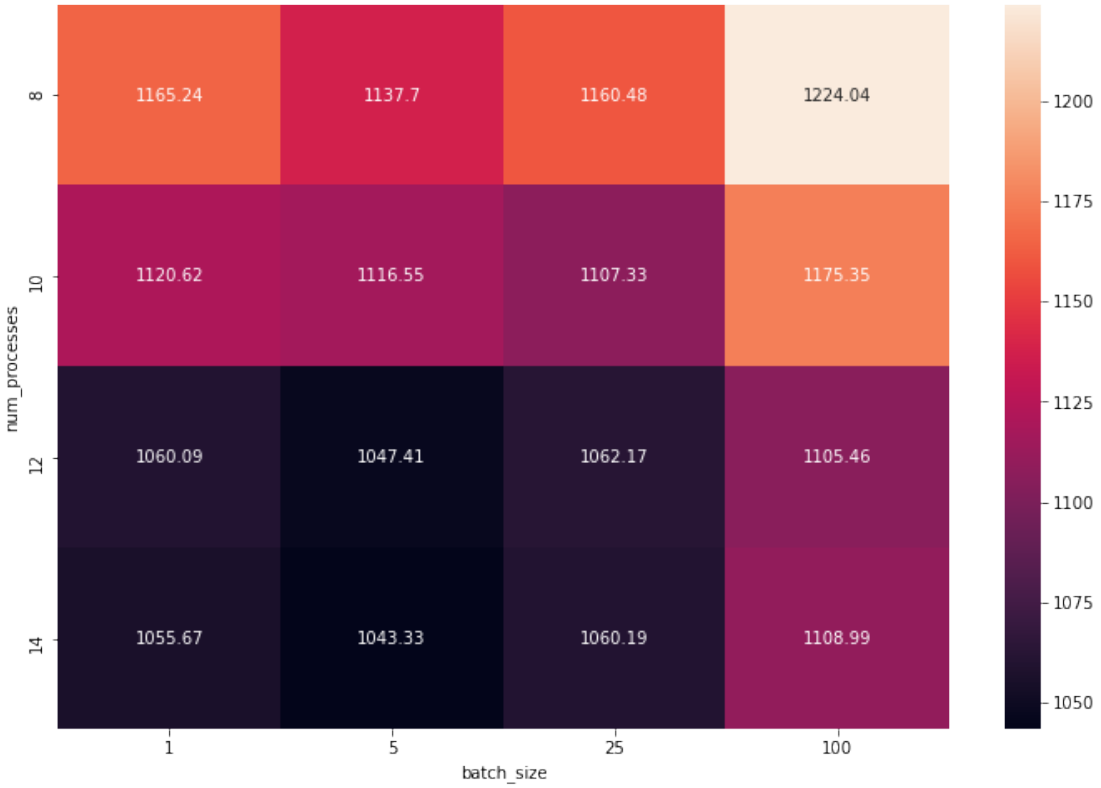


Figure 7.4: Batch v processes for small-scale optimal-solver experiment with database saving

Solvers support rectangular samples but these were not included in the experiment due to time constraints. They would largely expand the number of experiment parameter combinations. If a non-square sample size AxB is used, then it will be tempting to experiment with switching it to BxA, and maybe even allowing a sample to switch between both varieties within a turn. This would lead to increasing the experiment run time by a factor of 2-4.

7.3.2 Database

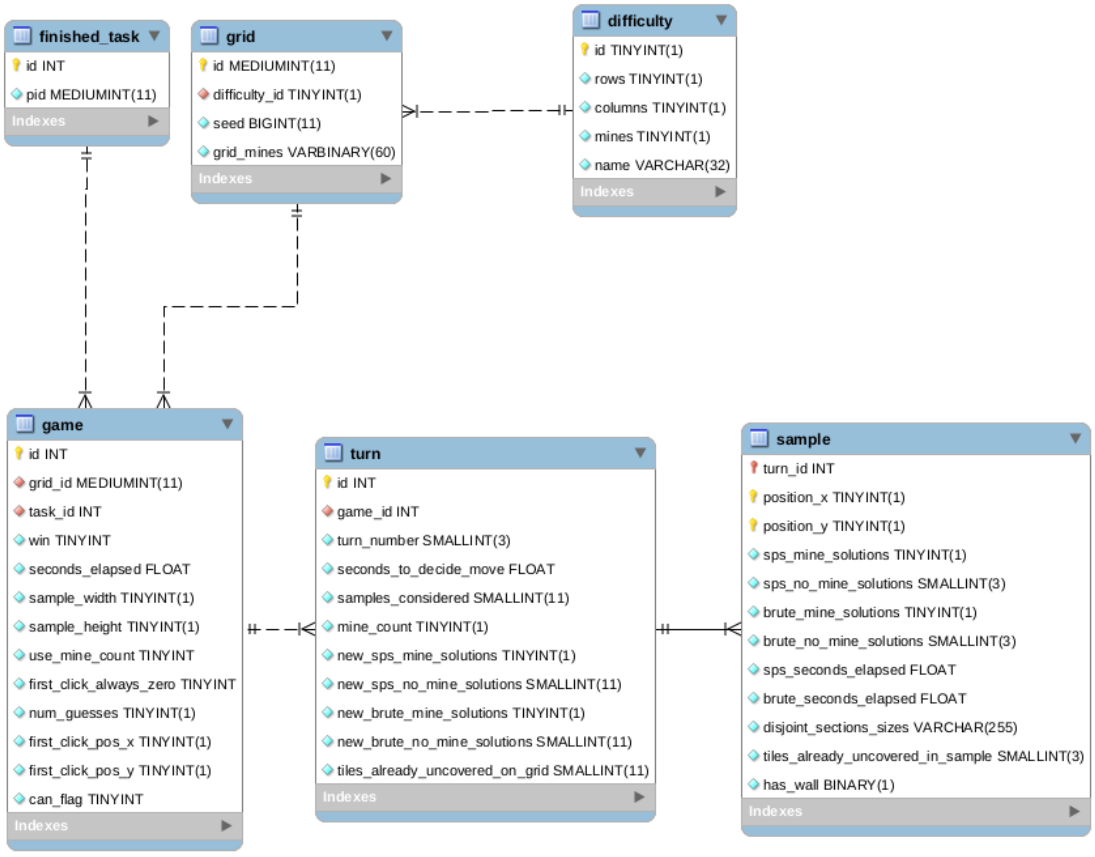


Figure 7.5: ER diagram of database model used for storing the data of the optimal-solver experiment

Data being collected from main experiment is deliberately detailed and structured, rather than being a collection of summary data. Summary data only allows for a limited amount of interesting questions to be explored. When collecting a richer data set, the approaches that can be taken towards the analysis increases dramatically. The purpose of this was that even the questions that weren't thought up of initially can still be explored; there wouldn't be a need to redo the experiment just to collect that particular required information.

A list of possibly interesting questions was composed alongside the information that would be required. A data model was built to try and include all this information in some way (either directly, or indirectly where this information can still be recovered/calculated somehow). Of course, as 100 million games are to be played, there would be a lot of structured data to store. The need for a database was clear. Certain bits of data were considered but not kept, such as the contents of the each sample considered.

Routine physical incremental backups were made on a separate hard-disk. In case something went wrong, better not to lose weeks worth of results and end up having little to show for by the end of the project deadline.

Physical backups better suited for when there is a very large data-set. As there are billions of entries here for the optimal-solver experiment, we went for physical backups. Incremental backups were made during run of experiment on an automatic schedule for convenience.

Experiment was estimated to have a size of about 120GB, actual memory size was 50GB for a complete run.

Every insertion of task's results to the database was done in a transaction so that in the case of a failure, we do not end up with partial data being stored leading to inconsistencies. It was crucial to uphold the data integrity, especially as the experiment would last many days.

7.3.3 Handling Interrupts

By including a finished_task table in the database, there was a way to keep track of which tasks were complete. On experiment start-up, the list of all finished tasks is queried and filtered out accordingly. Since every insertion of task results was done as a single transaction, if for whatever reason any bit of that data insertion failed then this would be rolled back and the task would not appear in the finished_task table. To complete the rest of the remaining unfinished tasks, we simply just restart the experiment script.

This ended up being a pretty robust implementation that coped with a dozen or more interrupts. The decision to include this tolerance to interrupts was a crucial decision that enabled the optimal solver experiment to be run to completion within the project's timeline.

8 Results and Discussion

In this section we explore some of the data from the experiments (Chapter 7), referring back to the stated hypotheses in Sub-Chapter 2.1. We discuss some of the conclusions that may be drawn.

8.1 Comparison of Solvers

The solvers we compare are the two implemented in this project (Chapter 5 and Chapter 6, and the two known best probabilistic solvers made by others (Ed Logg and David N. Hill, further details in Sub-Chapter 1.9).

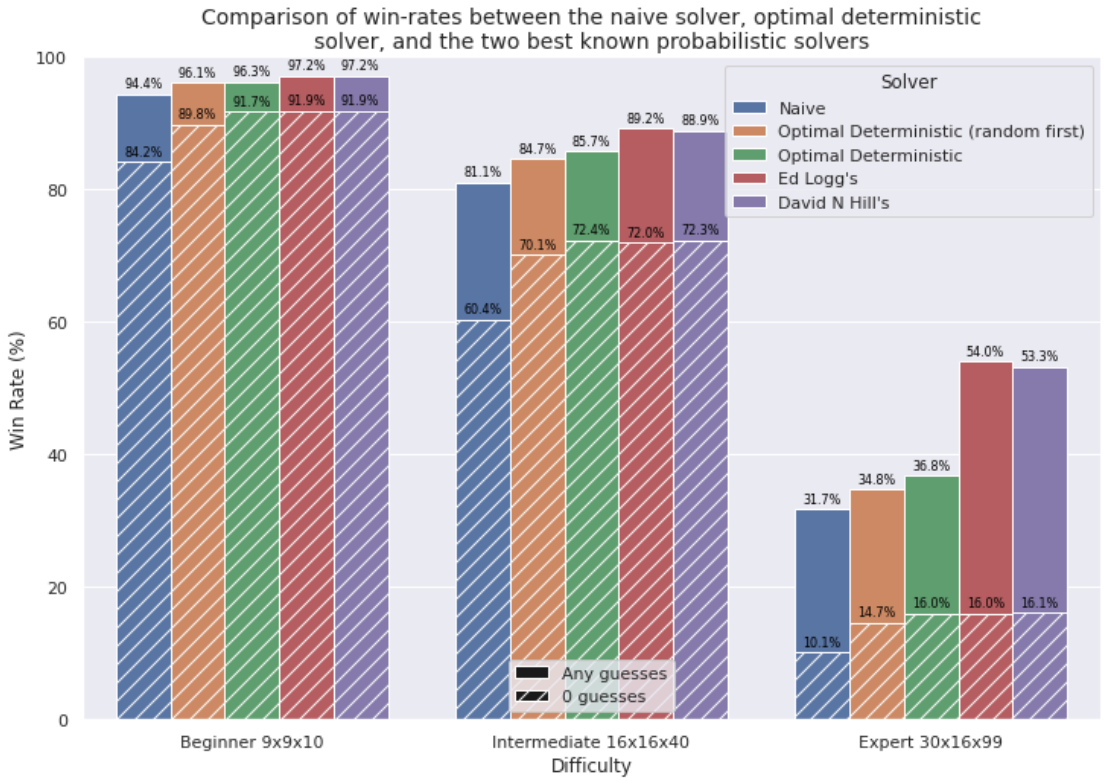


Figure 8.1: Win-rates of the naive, optimal deterministic, and the two known best external probabilistic solvers.

The win-rates in Figure 8.2 of the optimal deterministic solver with a fixed first-click position are those from the verification test run. The ones with random first-click are those from the main optimal deterministic experiment. Both use the same solver, however the latter was run on fewer games (n =15,000 as opposed to n=100,000) in the conditions that gave the shown win-rates.

FIRST-CLICK POSITIONS The first click positions are known to make an impact on the outcome of games[1, 6] . Listed in Table 8.1 are the first-click positions used by the solvers to get the win-rates seen in Figure 8.2. Note that the coordinates (x, y) of first click positions are 0-based, with the tile in the top-left corner of the board being $(0, 0)$.

Solver	Beginner	Intermediate	Expert
Naive	random	random	random
Optimal Deterministic (random first-click)	random	random	random
Optimal Deterministic	(3, 3)	(3, 3)	(3, 3)
Ed Logg's	(2, 2)	(3, 2)	(3, 3)
David N Hill's	(2, 2)	(3, 3)	(3, 3)

Table 8.1: First-click positions used by the solvers shown in Figure 8.2

The impact of fixing the first-click position is a bit larger for the 0-guess win-rates than for the overall win-rates on the easier difficulties, implying a poor choice in first-click position increases the probability of having to make a guess but this is more costly for Expert boards than it is for the others.

OPTIMALITY OF SOLVERS It can be seen in Figure 8.2 that the 0-guess win-rates match those of the external solvers, which provides strong evidence that the optimal solver is indeed optimal for deterministic plays, or at the very least, near-optimal. This gives credence to the reliability of the win-rates achieved by the claimed optimal deterministic solver and are shown and discussed throughout this chapter.

Furthermore, it seems convincing that these are indeed the best possible win rates when playing optimally on games that can be solved without guessing (with sensible first-click position choices) as there are now at least 3 independant solvers whose maximal win rates are very close to one another for these types of games. The slight discrepancies between them may be owing to the slight differences in first click positions used and/or be the result of chance.

RANDOM GUESSES VS PROBABILISTIC PLAY The benefits of making even a random guess to a deterministic play has a huge impact on the win-rates, especially for the more difficult boards. This contradicts Hypothesis 6. On expert boards, proportion of boards won more than doubles when the agent is allowed to guess. Although a deterministic approach with random guessing is not far from a good probabilistic play on beginner boards, there is a significant gap between them on intermediate and expert boards. It seems there is great utility in figuring out what the better guesses are if Expert boards are being played, although a deterministic player can still perform relatively well nonetheless.

NAIVE SOLVER In direct contradiction to Hypothesis 7, the naive solver (Chapter 6) performs remarkably well in comparison to the other solvers. The naive algorithm is a deterministic algorithm that tries to replicate an intuitive strategy a human would use. This result suggests that the approach of a human player seems to be already close to optimal (for deterministic players), despite it ignoring the sorts of cases which require brute-force searching through

all possible mine configurations which are the kinds of problem instances characteristic of NP-hard and coNP-hard problems. This is discussed in more detail in 8.3, along with the naive solver's other results.

8.2 Optimal Solver Results

Although the planned number of games for the experiment was 25,000 per experiment parameter combination, the experiment was stopped early at 15,000 due to time constraints. This may reduce the precision of the results slightly. Based on the previously observed convergence of win-rates, a small absolute error as large as 1% in the win-rates may be expected.

8.2.1 Win Rates

Here we instead compare various views of the overall results, sometimes deliberately not extracting out particular variables to emphasise the differences in win-rates that variable causes, seen as a thin black strip on a bar, when all the other variables are fixed to a specific value. The top and bottoms of the black strips, such as those seen in Figure 8.3, indicate the upper and lower win-rates from setting the variable to true or false, respectively.

Refer to the Appendix for a more complete series of charts showing all the win-rates across all of the experiment parameter combinations.

Note that a gap in a group of sample sizes for Beginner difficulties signifies the sample sizes that were skipped in the experiment as they are large enough to contain the entire grid. We specifically label the whole-grid sample sizes and colour-code them for clarity.

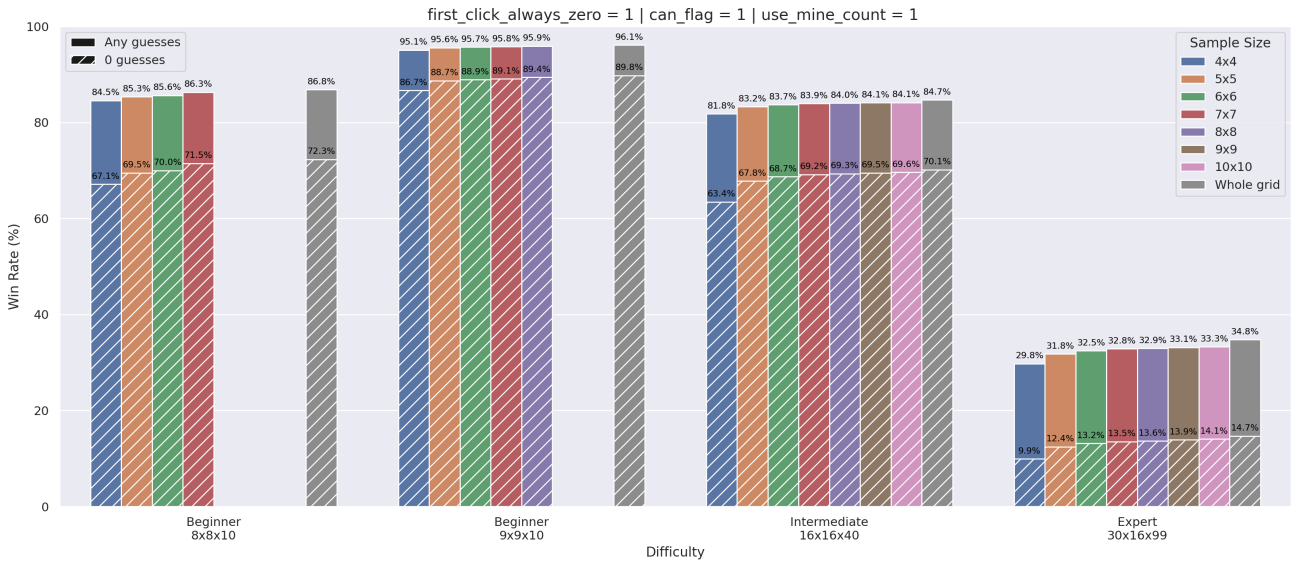


Figure 8.2: Optimal deterministic solver win-rates under favourable conditions

Figure 8.2 shows the best possible win rates of the deterministic optimal solver (Chapter 5)

across sample sizes when it is given the most favourable conditions. Unsurprisingly, the solver performs better on a more modern 9x9x10 beginner board than the old-school 8x8x10 board due to the decreased mine density of the board.

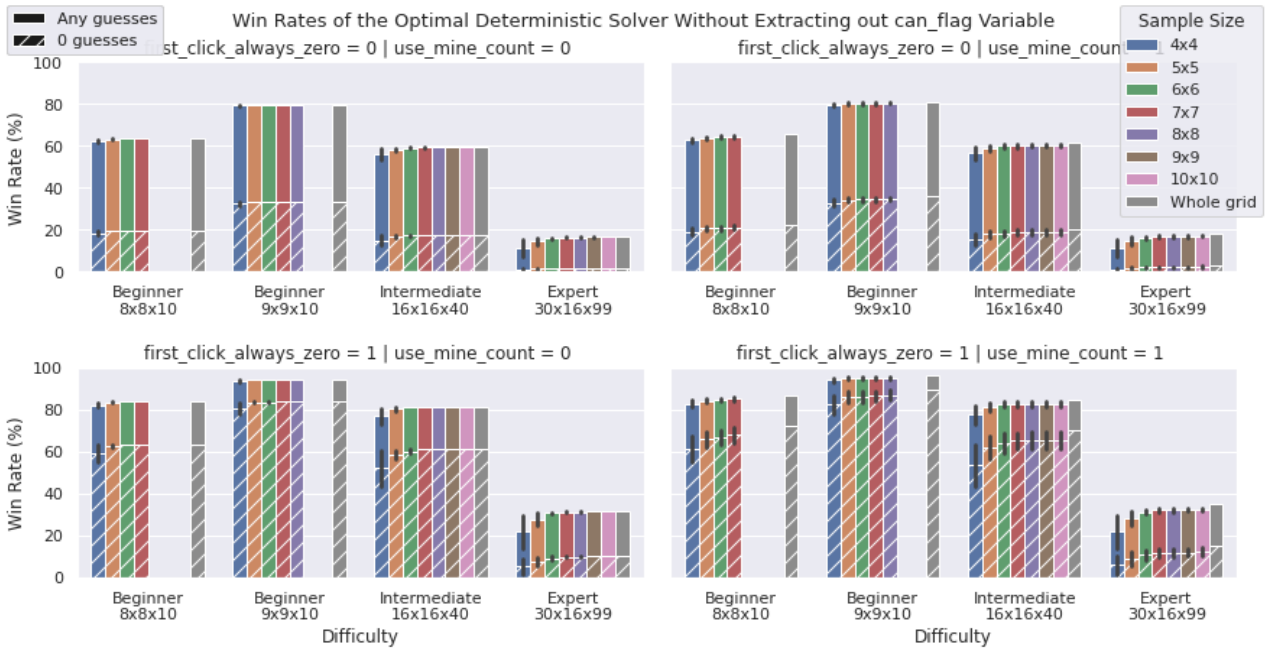


Figure 8.3: Optimal solver win rates, emphasising the impact of using flags

IMPACT OF AGENT'S ABILITY TO FLAG TILES It can be seen from Figure 8.3 that the use of flagging tiles becomes more valuable for smaller sample sizes, and for increasing difficulties.

When mine count is ignored by the solver, the utility of flags diminishes to nothing if the sample size is large enough and/or the difficulty is easy enough. By including the mine count in the decision making, suddenly flagging becomes useful throughout all expect the whole-grid samples.

Consider that the benefit in using flags is in acting as a mechanism which propagates information across samples. This result then suggests that solving adjacency constraints, i.e., solving a tile on the numbers displayed on the board, is largely local. To put it another way, the information necessary to determine a solution to a tile based on the board's state can often be found in that tile's surrounding area; you usually do not have to consider the other side of the board.

A sample size of about 5x5 or 6x6 seems plenty for beginner and intermediate boards and covers most cases for expert too, although for the latter there still is a marginal advantage to going as far high as 9x9 or even 10x10 samples.

Despite this result, it is worth noting that there are exceptions to this as you can always construct scenarios where the whole grid must be considered to solve it, otherwise this would not be a coNP-hard problem.

When also considering the mine count, it seems that this transmission of information is often useful. This is probably because the flags act as a memory between samples which

allows for a better estimate of the range of mines that must be contained within the frontier tiles of the sample.

Interestingly, the increase in win-rate in this case seems to remain stable throughout the larger sample sizes; even the larger sample sizes benefit from using the mine count. This is probably due to the fact that flagging a tile changes the mine count for all samples and so its information that is transmitted globally, regardless of sample size. Even having a sample that covers almost all of the grid will have a wide range of possible mines it could contain because of the unchanging mine count, causing it to miss solutions.

As predicted by Hypothesis 5, if the optimal solver can see the whole grid then the ability to flag has no effect on the win rate. The exact wins were checked to ensure the lack of the black strips was not due to a tiny (but non-zero) difference in wins; they stayed exactly the same between being able to flag and not. Going back to the idea of flags acting as a mechanism (or a sort of memory) for passing information across samples, it is easy to see why whole-grid samples do not benefit from it. There are no other samples to pass information to; it's all there.

A significant increase in the differences of win-rates can be seen when switching between whether first-click is always zero, or sometimes. This however may be due to a proportional increase in the win rates achieved in general exacerbating the benefit of using flags.

The impact of using flags seems to be generally greater on the 0-guess win-rates than the overall win-rates, especially on intermediate boards. Perhaps by using flags the solver needs to guess considerably less often, and so more games begin to fall under the 0-guess category. These winning games still contribute to the overall win-rate, but now a greater proportion of them require no guesses at all.

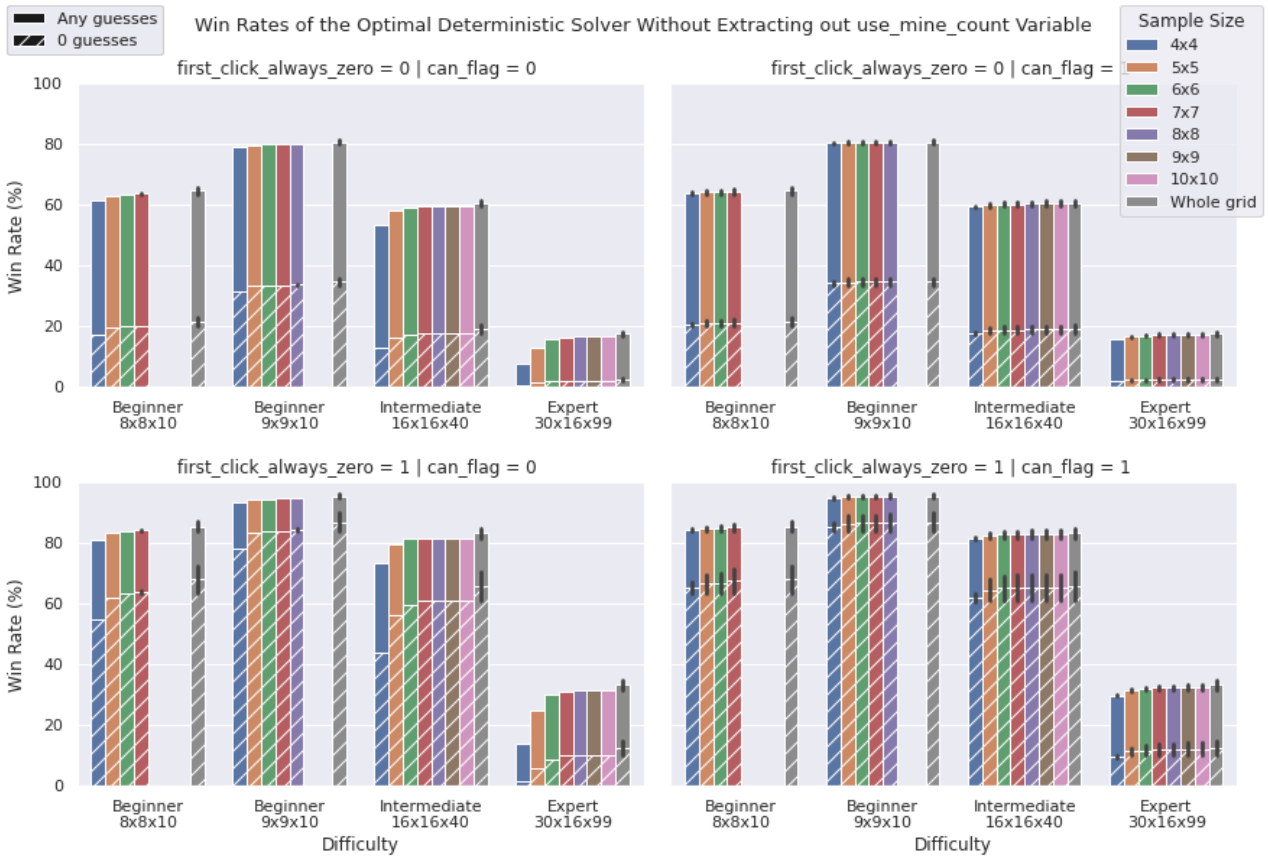


Figure 8.4: Optimal solver win rates, emphasising the impact of using mine count

IMPACT OF USING THE MINE COUNT We can see that there is a slight gradual increase as sample sizes increases when mine count is used. This upwards trend in win-rates is absent when the mine count is ignored by the solver, with the win-rates flattening out once a large enough sample size is reached for a difficulty. This is further evidence of the locality of Minesweeper.

Figure 8.4 shows that the use of mine constraints has a small impact on the overall win rate for when first-click is not guaranteed to be a zero. When the first-click is always zero, the inclusion of mine count makes a more significant difference.

When the solver cannot flag tiles, the inclusion of the mine count only really helps when the sample size covers all, or almost all, of the board. When the solver can flag, the mine count can help out for almost all sample sizes. An explanation of this, as mentioned under Figure 8.3, is that the lack of ability to flag tiles means the global mine count cannot change. Figure 8.4 further implies that this unchanging total mine count causes the mine constraint to never be able to solve enough of these cases when the sample size is too small; the mine constraint does not enforce a strict enough mine count range to extract the solutions.

From this we conclude that it does not make sense to ever consider the mine count unless

either we can also flag, or at least the vast majority of the board is visible.

It is clear from this that a well-performing deterministic agent can readily ignore the total mine constraint without sacrificing much in its ability to solve boards. Perhaps this is because using the mine count comes in handy only in end-game situations, and even then, their necessity in the end-game is relatively rare.

An interesting thing to note is that the benefits that of using the mine count is substantially increased for 0-guess games by the use of the first-click-is-always-zero rule. This may simply be due to the accompanying major increase in win-rates when using that rule, which just means that more games are entering the end-game scenarios to actually be able to make use of the mine count to solve samples.

There is a gradual but noticeable increase in the benefits of using mine constraints as sample size increases. This is expected as the mine count constraint is global, not local; having as much information as possible about the number of non-flagged closed tiles remaining across the board leads to tighter bounds on the constraint, which in turn, leads to finding more solutions and thus winning more games.

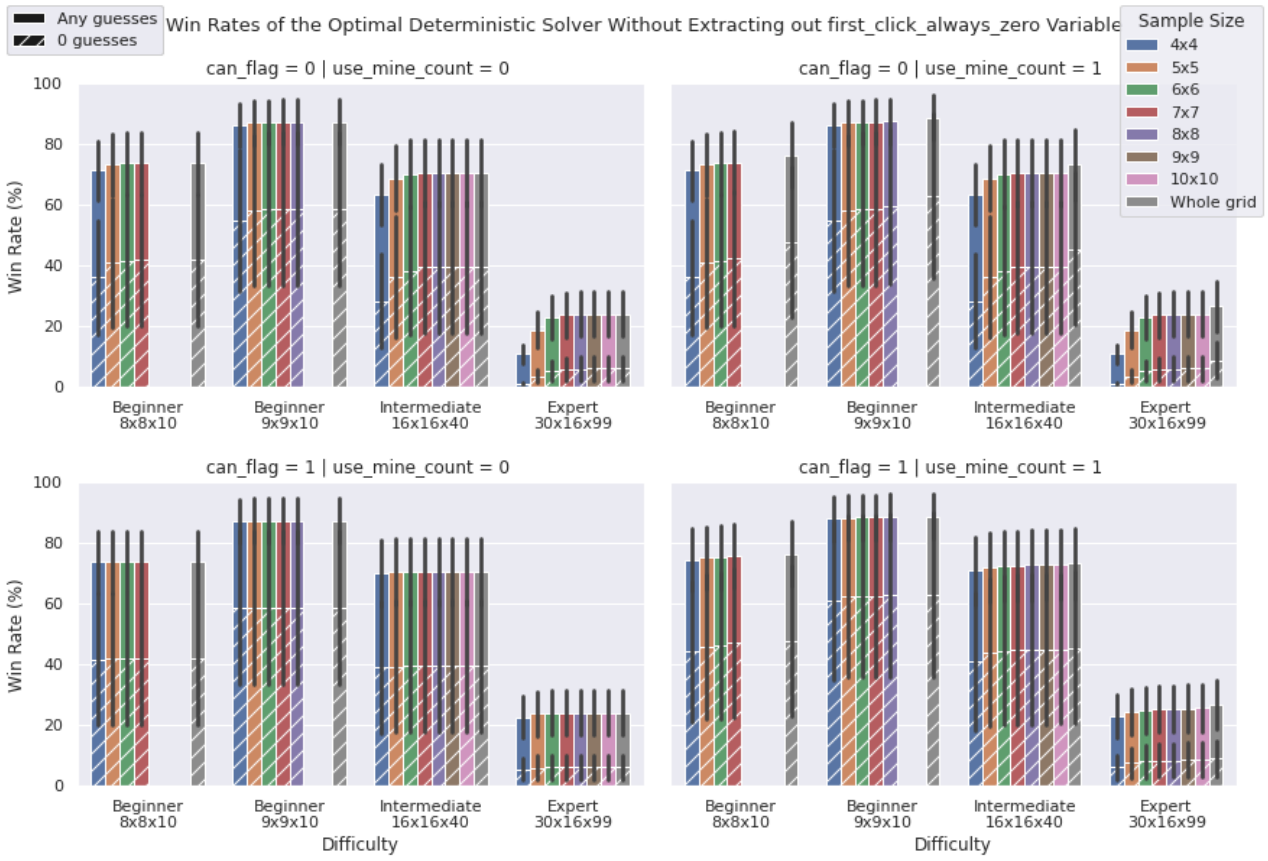


Figure 8.5: Optimal solver win rates, emphasising the impact of first-click-always-zero rule

IMPACT OF FIRST-CLICK-ALWAYS-ZERO RULE From Figure 8.5 it is clear that the difference in the specifics of the safe first-click rule makes a big difference across all areas. Older versions of Minesweeper (before 2007 Windows Vista version[13]) do not guarantee that the first click is always a zero, just that it is safe, and so you should expect the older versions of Minesweeper to be less winnable too. This is an important factor to consider when looking at any results regarding wins in Minesweeper, as the difference in the rules around the version of Minesweeper can make a substantial difference. Additionally, if you share any results regarding Minesweeper, it is also a good idea to specify the version of Minesweeper version used, more precisely, whether the first-click is guaranteed to be zero or not.

The difference in win-rates seems to be larger for 0-guess games than for overall win rates on beginner and intermediate boards.

The query results shown in Figure 8.6 show the time taken to run each individual game, added up across all batched tasks. The figures equate to 4.3

name	total_seconds_elapsed	avg_seconds_elapsed	num_games
Beginner (pre Windows 2000)	367240.5013473034	0.6120675022455057	600000
Beginner	356285.07285261154	0.4948403789619605	720000
Intermediate	2925868.6017723083	3.047779793512821	960000
Expert	10335295.329180002	10.765932634562501	960000

Figure 8.6: Total and average serialised times of games played in the optimal solver experiment

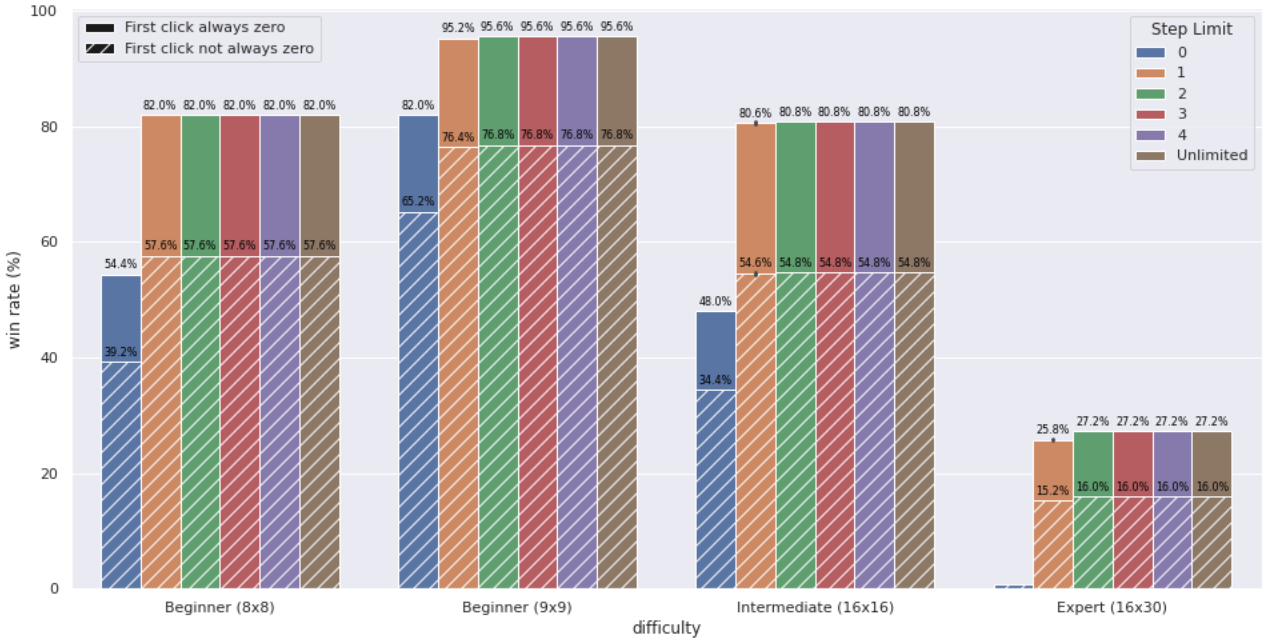


Figure 8.7: n-step naive algorithm's win-rates across different board difficulties

8.3 Naive Solver Results

A criticism for the naive algorithm could be that it's iterative and exhaustive approach arguably imitates an extremely diligent human player. A human player would arguably play more akin to the 1-step or 2-step versions (where 0-step is just the single-point strategy). The results in Figure 8.7 show that even with limitation, the solver performs very well. In-fact, even 1-step covers the vast majority of cases that such a strategy can solve, which encompass many of the common patterns that are seen when playing Minesweeper, and 2-step seems to perform just as well as the unlimited exhaustive version. These limited versions can be implemented with an efficient polynomial algorithm and perform well.

This may seem a counter-intuitive result as solving each turn of Minesweeper is a co-NP problem, while this result shows that in-fact a simple and intuitive strategy is plenty enough to solve the majority of its instances. From this we gather that, despite Minesweeper's proven hardness result, it is easy in the average case. This makes sense when we consider the fact

that Minesweeper is a game that is supposed to be beatable, and so we would expect that the average player should be able to solve it without having to frequently resort to enumerating all possible mine configurations just to solve a typical game.

```
Preparing experiment 'Naive algorithm experiment':
Creating tasks... DONE
Experiment ready to run.


Running 25000 games for each of 96 different parameter combinations...

Total games: 2400000    Batch size: 5          Total tasks: 480000            Num processes: 12
100%|                                                                        | 480000/480000 [5:35:27<00:00, 23.85it/s]
```

Figure 8.8: Summary experiment information for the naive solver experiment

Figure 8.8 shows some summary data of the naive solver experiment, including that its actual run time was a consecutive 5 hours 35 minutes total.

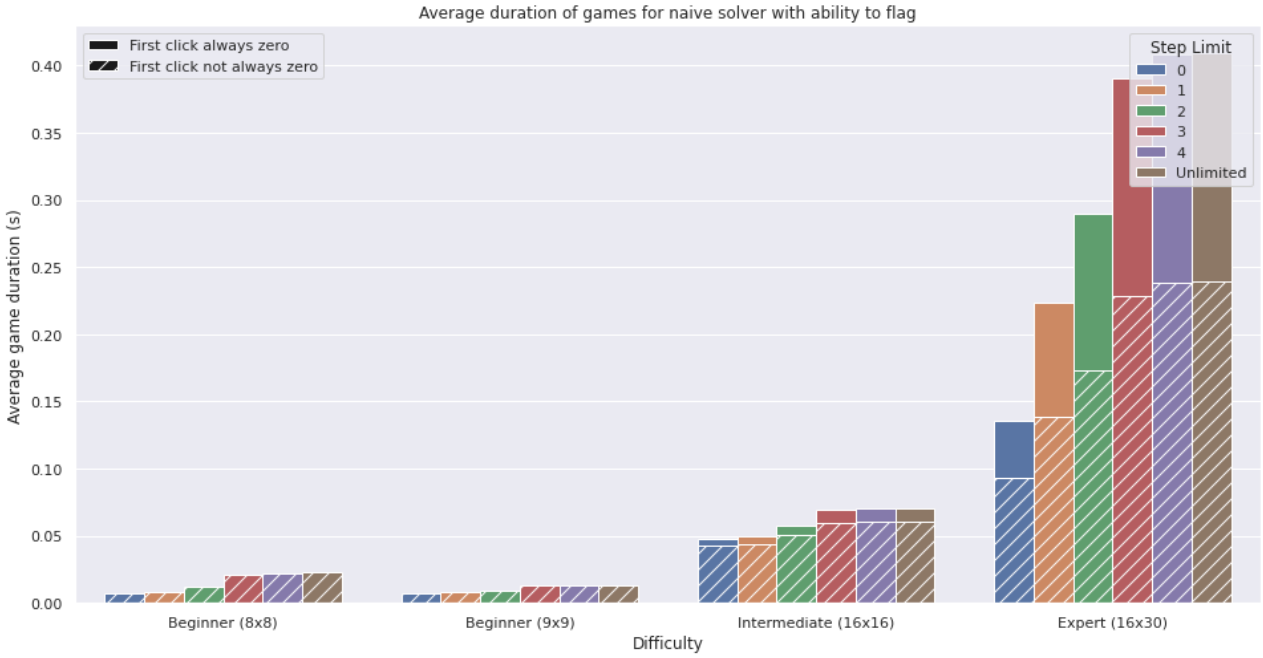


Figure 8.9: Average game times for naive solver with ability to flag

We can see from Figure 8.11 and Figure **??** Figure that having a higher sample size and harder difficulty increases the win-rate, which is a very predictable result. There is also a sizeable increase when the agent is provided the ability to flag tiles.

The increase in times seems to climb rather quickly as the difficulties get harder. A difficulty increase leads to a corresponding grid size increase, this could imply that the algorithm in its

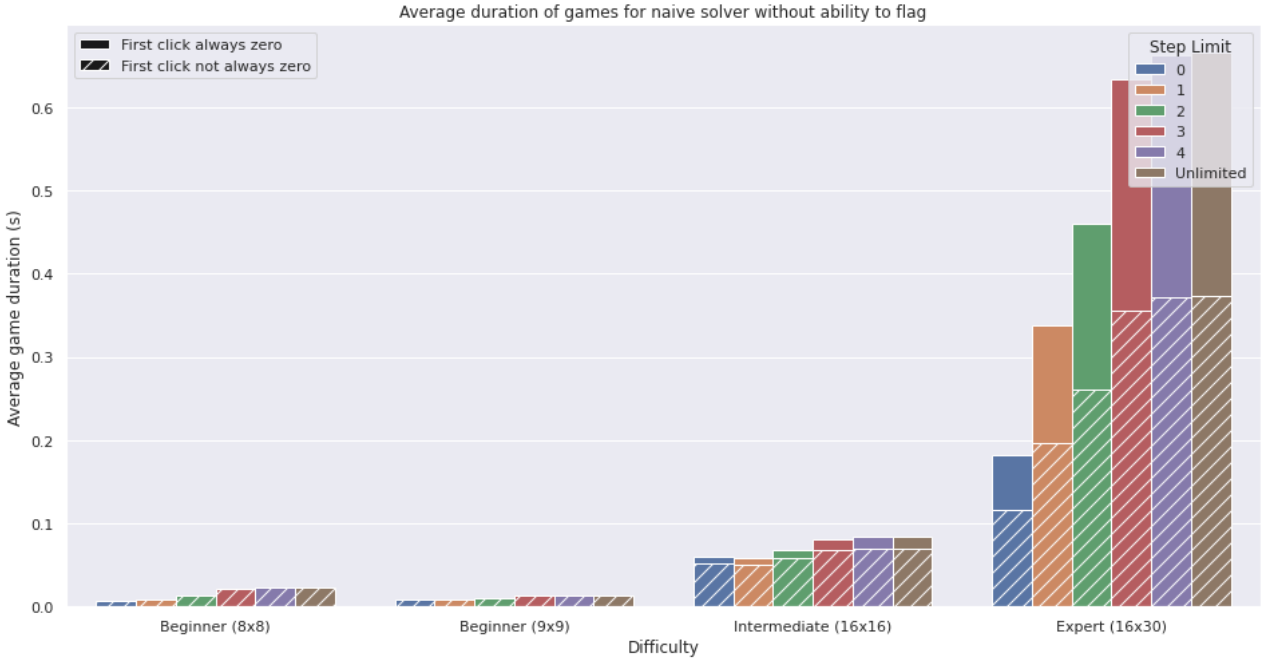


Figure 8.10: Average game times for naive solver without ability to flag

current form has a not-so-friendly time complexity that quickly ramps up as the input size gets bigger.

However, given the relatively short run time of the experiment, it is considerably faster than the optimal deterministic solver.

The number of samples considered seems to be mostly the same throughout most steps except for the 0-step version. For all difficulties except the expert boards, a step limit of 0 considers more samples than the other step limits. Moving onto the expert board gives us the reverse scenario: 0-step naive algorithm has significantly less samples that it considers than the others.

This reversal makes sense when we also look at Figure 8.7 and remind ourselves that the 0-step naive algorithm performs extremely poorly on expert board; the lower sample count can be explained by the fact that it is losing games much earlier than when the step limit is non-zero.

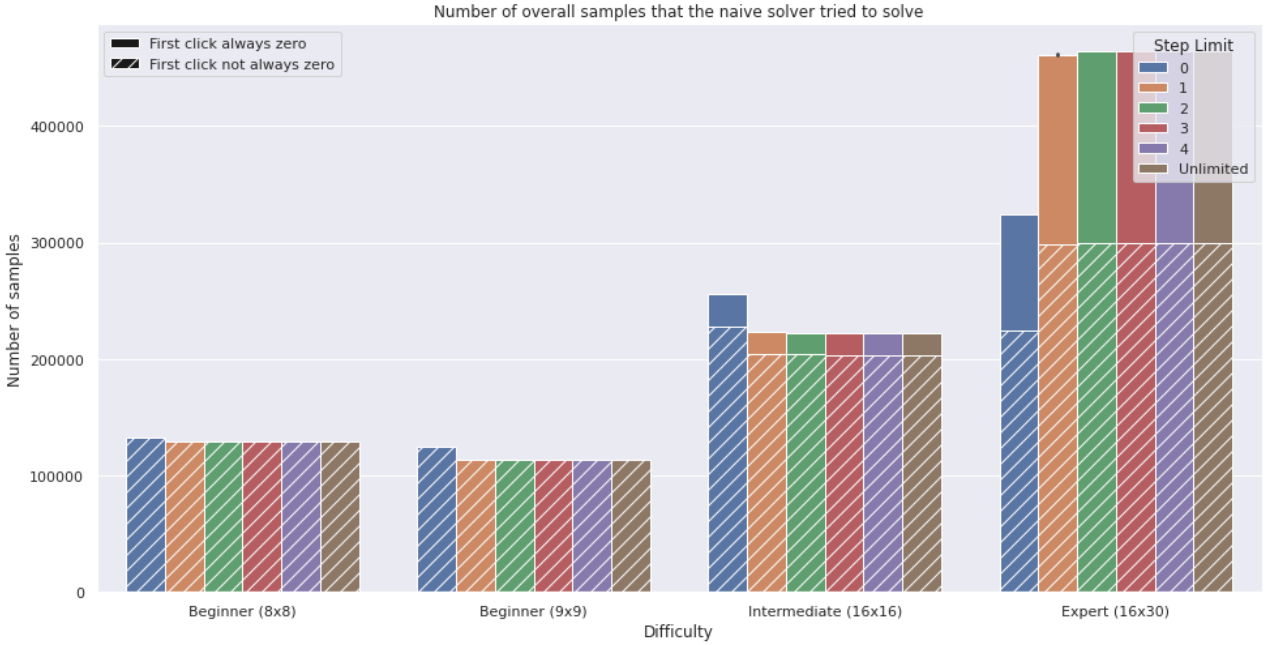


Figure 8.11: Overall quantity of samples considered by the naive solver

9 Conclusion

9.1 Hypotheses

The optimal deterministic solver performs pretty well when allowed to make a random guess, even when compared to the two best known probabilistic solvers which try to make the best guesses. Imposing rather strict informational restrictions on the optimal deterministic solver does not diminish its win rates greatly, until a certain threshold is reached (around a 5x5 sample size, below which a drastic dip in win-rates is seen across all configurations). This suggests that a 5x5 - 7x7 area contains enough information to win most games that a deterministic approach could hope to win.

The inclusion of mine count has a relatively small, but appreciable impact on the win-rate. This shows that it seems to be okay to design solvers which learn/solve Minesweeper based on a restricted view of the board without necessarily sacrificing too much on the potential performance. This also shows that, for adjacent-mines constraints, the effect of moves and decisions made are mostly local in Minesweeper, i.e., in most cases you do not have to consider what the other side of the board looks like to make a decision. However once you start to use the mine count too, looking at a larger portion of the board does become more necessary to reap the benefits of using the mine count.

The majority of win-rate percentage can be achieved through the deterministic approach (especially for Beginner and Intermediate grids), and so going for a purely deterministic solver and ignoring the probabilistic side is not a bad design decision unless optimal (or near opti-

mal) performance is desired. Including the probabilistic approach still would likely lead to a significant win-rate increase (especially on Expert grids) nonetheless.

The naive algorithm performed surprisingly well, considering that it is a polynomial algorithm and playing Minesweeper is a coNP-complete problem. This suggests that a solver based around the use of such an algorithm, something similar, can perform remarkably well without needing to worry about the more complicated and cumbersome of brute-force enumeration of all possible mine configurations. For example, for creating a Minesweeper learner, this result suggests that the rule inductive approach may in-fact be a promising one, especially since there already exists a learner that has learnt the very same style of rules as the naive algorithm (just not to an extensive enough state to be able to perform well however).

9.2 General Remarks

The solvers that were sought after were implemented, and the questions asked were investigated. A variety of relevant conclusions have been drawn up and so this has been mostly a successful project.

However, with the time pressure of the deadline, there has been a significant drawback on the quality of the report, and a lot of potential analyses have never had the chance to be done. Being unable to properly present the work is what I would consider to be the partial failure to this otherwise generally complete project.

Throughout the project, there have been some key decisions and habits that have caused either great strides in progress, or ended up being productivity pitfalls.

Unfortunately, too many significant delays to critical tasks (such as launching the optimal-solver experiment) have been made within the project's timeline. These delays caused setbacks in other important areas of the project, such as report writing, experiment data analysis, and even the number of games that could be played for the experiments (optimal solver experiment was stopped early).

There were some key decisions that greatly improved on the project (or prevented things from getting much worse), such as the use of the CP-SAT solver, and the reliance on web-based tools which saved all of my project-related data on the cloud (preventing an event where my home computer's hard drive to be wiped from having any real consequences on the project).

The use of proper multiprocessing and creating a fault-tolerant experiment running environment for the optimal-solver experiment both proved to be vital in managing to get a substantial amount of the data ready in time for some analysis to be done.

REFERENCES

[1] D.J. Becerra. "Algorithmic Approaches to Playing Minesweeper". Harvard College, 2015.

[2] R. Collet. *Playing the Minesweeper with Constraints*. DOI: https://doi.org/10.1007/978-3-540-31845-3_21.

[3] Google. *CP-SAT Solver*. URL: https://developers.google.com/optimization/cp/cp_solver.

[4] D.N. Hill. *Minesweeper*. GitHub repository for one of the best solvers. URL: https://github.com/DavidNHill/Minesweeper.

[5] R. Kaye. "Minesweeper is NP-complete". In: *Mathematical Intelligencer* 22.2 (2000), pp. 9–15.

[6] E. Logg. *MineSweeper solver*. GitHub repository for one of the best solvers. URL: https://github.com/EdLogg/MineSweeper.

[7] E. Logg. *Odds of winning at Minesweeper*. Discussion thread where Ed Logg and David N. Hill (BinaryChop) share their solvers. URL: https://www.reddit.com/r/Minesweeper/comments/8b3b30/odds_of_winning_at_minesweeper.

[8] R. Massaioli. *Solving Minesweeper with Matrices*. URL: https://massaioli.wordpress.com/2013/01/12/solving-minesweeper-with-matricies.

[9] Lourdes Peña-Castillo and Stefan Wrobel. "Learning Minesweeper with Multirelational Learning". In: Jan. 2003, pp. 533–540.

[10] quantum_p. *How to Win at Minesweeper*. URL: https://quantum-p.livejournal.com/19616.html.

[11] A. Scott, U. Stege, and I.V. Rooij. "Minesweeper May Not Be NP-Complete but Is Hard Nonetheless". In: *The Mathematical Intelligencer* 33 (2011), pp. 5–17. DOI: 10.1007/s00283-011-9256-x.

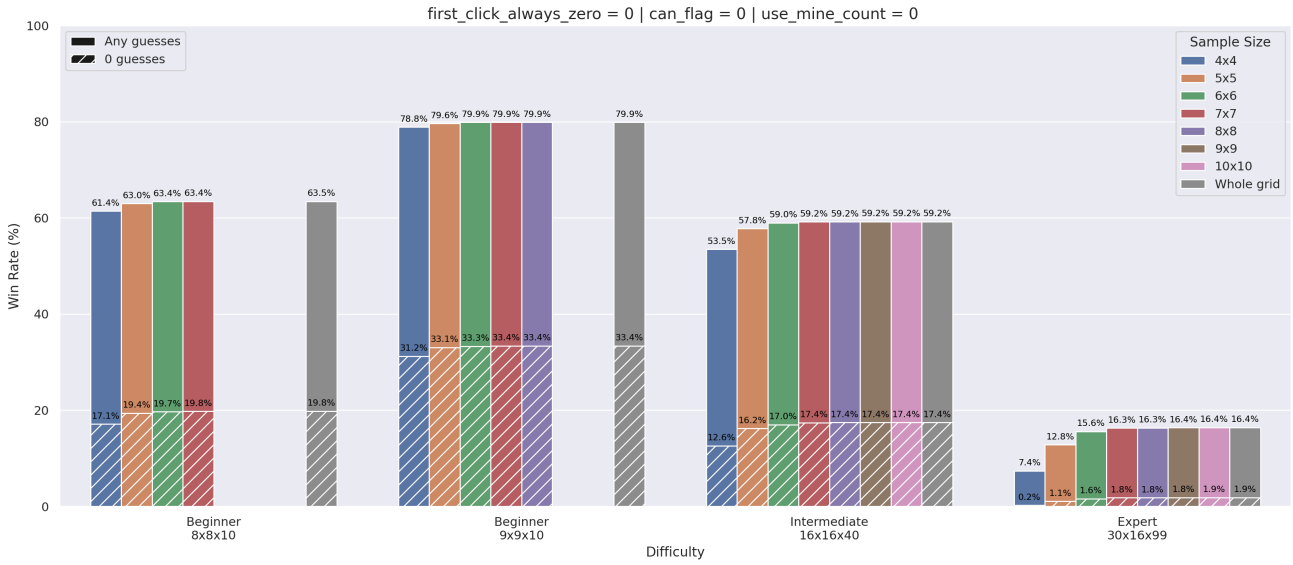
[12] *TimeComplexity*. Python Wiki. URL: https://wiki.python.org/moin/TimeComplexity.

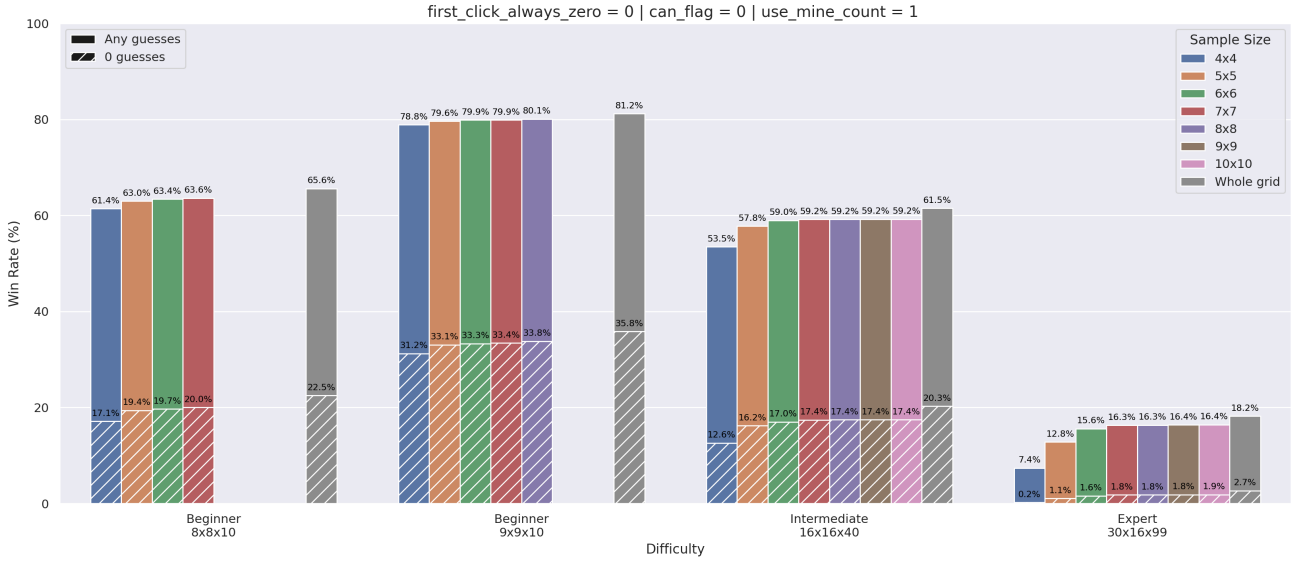
[13] Minesweeper wiki. *Windows Minesweeper*. URL: www.minesweeper.info/wiki/Windows_Minesweeper.

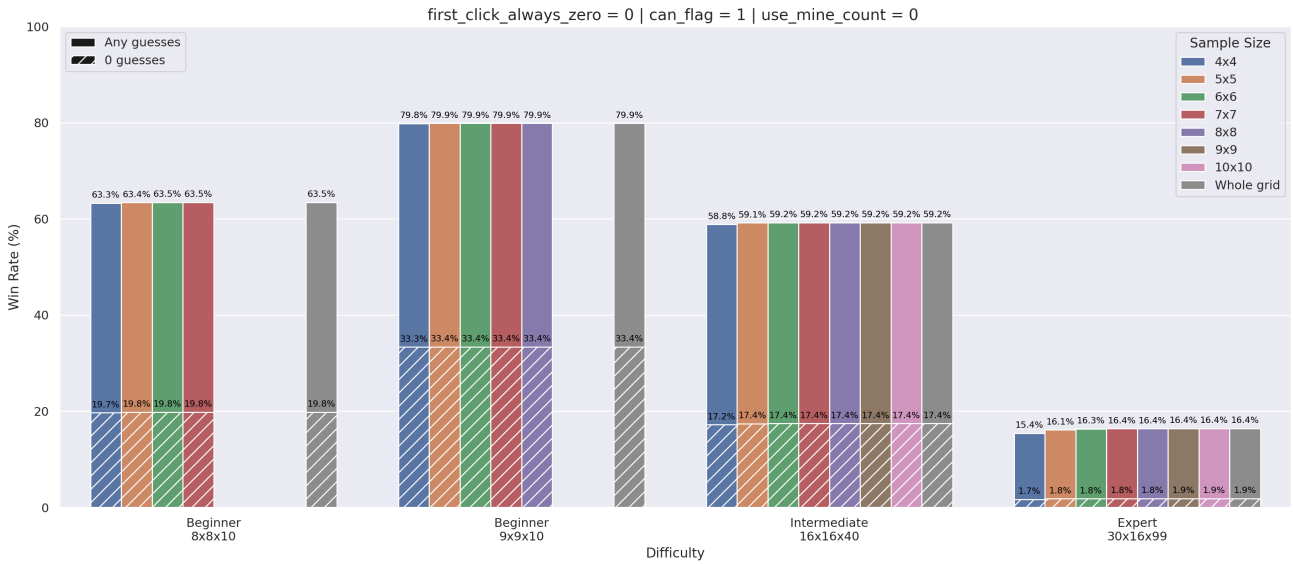
APPENDIX

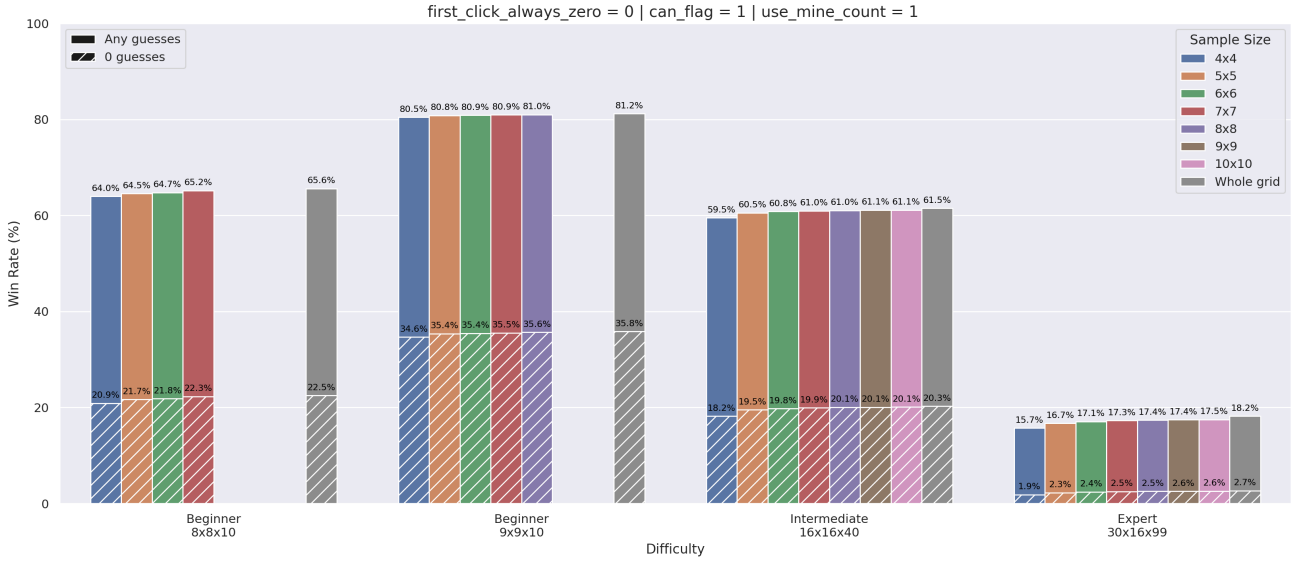
RESULTS EXTRA

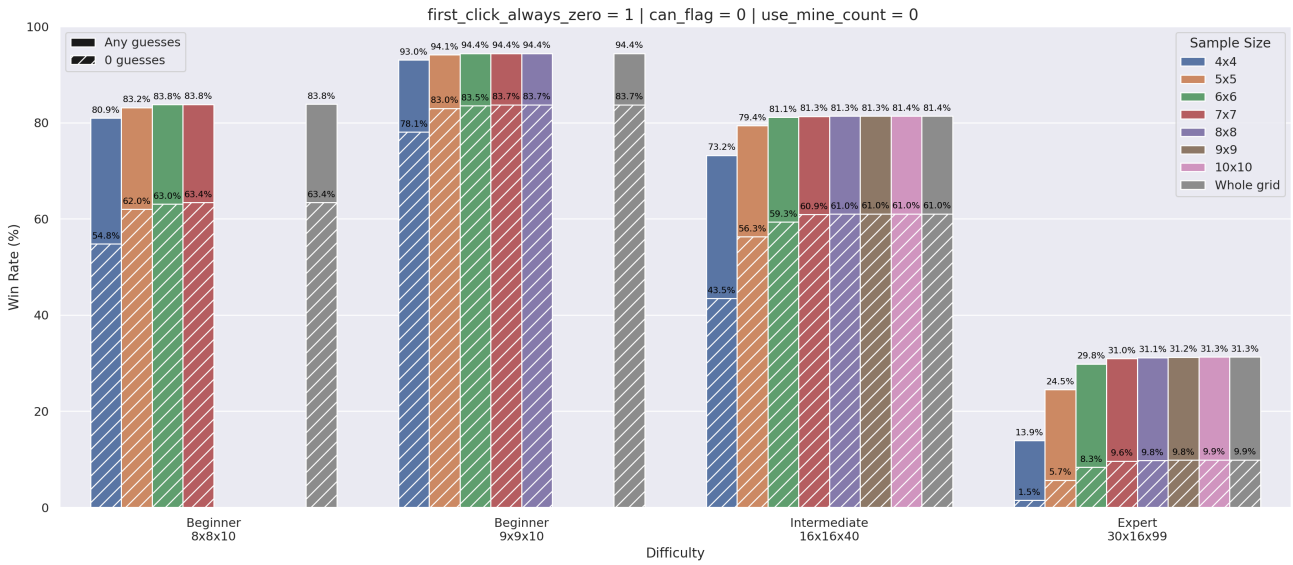
Listed below are 8 figures that display the entirety of the main win-rate results for the optimal deterministic solver experiment. For a further analysis and discussion on the data, see Sub-Chapter 8.2

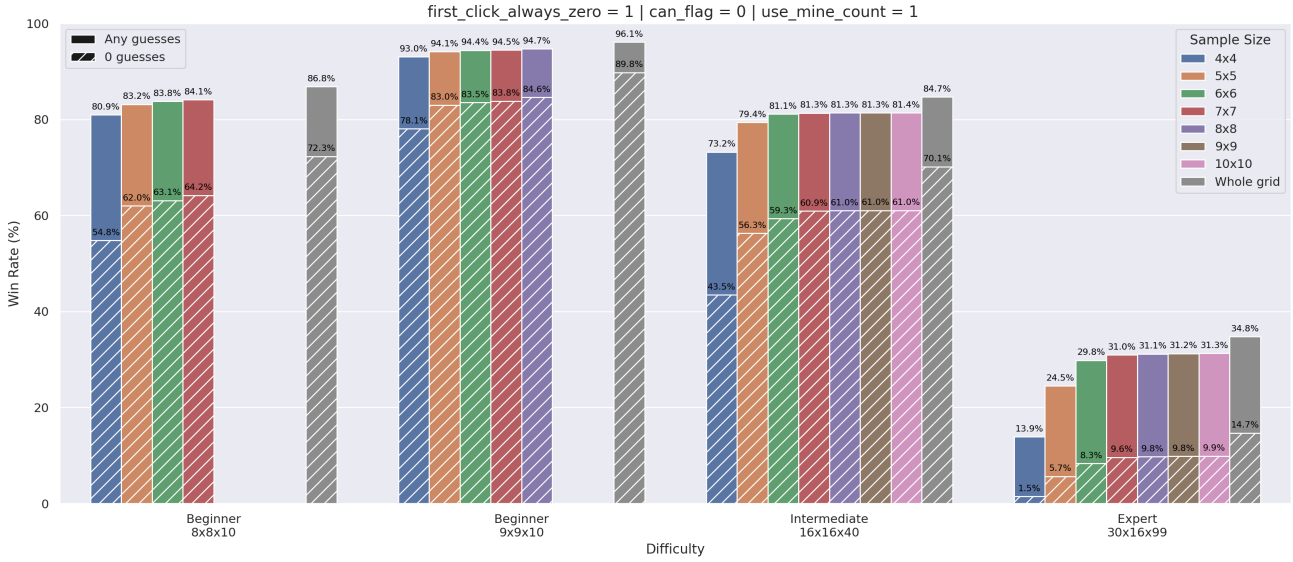


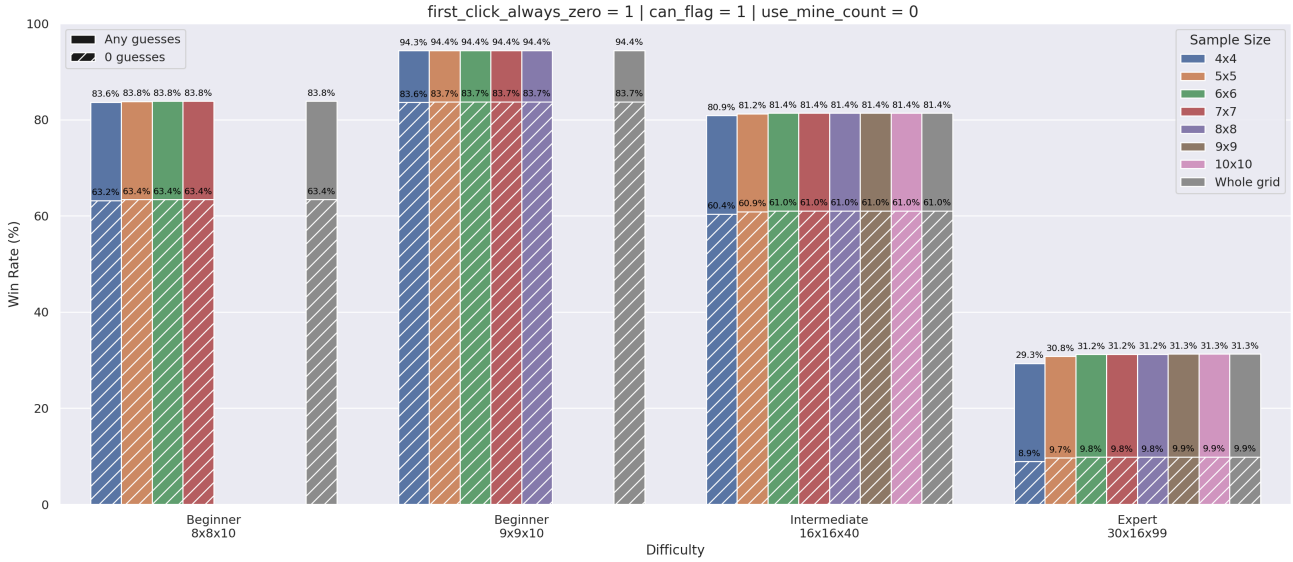

first_click_always_zero = 0 | can_flag = 0 | use_mine_count = 1
Any guesses
0 guesses
Sample Size
4x4
5x5
6x6
7x7
8x8
9x9
10x10
Whole grid
Win Rate (%)
Beginner 8x8x10
Beginner 9x9x10
Intermediate 16x16x40
Expert 30x16x99
Difficulty

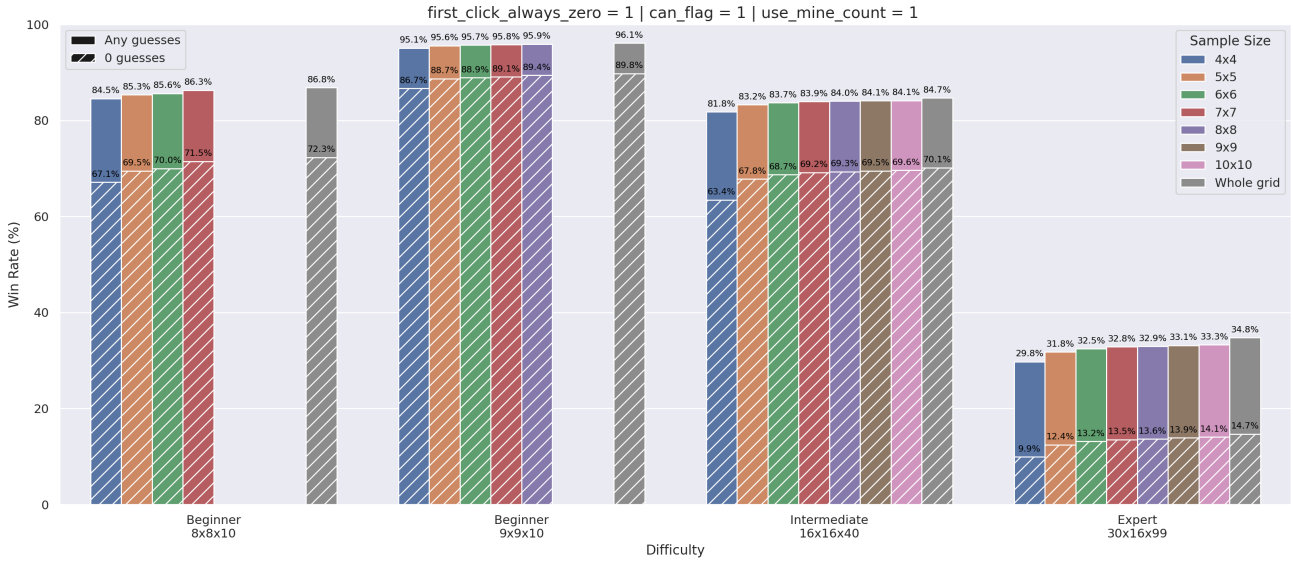

first_click_always_zero = 0 | can_flag = 1 | use_mine_count = 0
Any guesses
0 guesses
Sample Size
4x4
5x5
6x6
7x7
8x8
9x9
10x10
Whole grid
Win Rate (%)
Beginner 8x8x10
Beginner 9x9x10
Intermediate 16x16x40
Expert 30x16x99
Difficulty


first_click_always_zero = 0 | can_flag = 1 | use_mine_count = 1
Any guesses
0 guesses
Sample Size
4x4
5x5
6x6
7x7
8x8
9x9
10x10
Whole grid
Win Rate (%)
Beginner 8x8x10
Beginner 9x9x10
Intermediate 16x16x40
Expert 30x16x99
Difficulty


first_click_always_zero = 1 | can_flag = 0 | use_mine_count = 0
Any guesses
0 guesses
Sample Size
4x4
5x5
6x6
7x7
8x8
9x9
10x10
Whole grid
Win Rate (%)
Beginner 8x8x10
Beginner 9x9x10
Intermediate 16x16x40
Expert 30x16x99
Difficulty


first_click_always_zero = 1 | can_flag = 0 | use_mine_count = 1
Any guesses
0 guesses
Sample Size
4x4
5x5
6x6
7x7
8x8
9x9
10x10
Whole grid
Win Rate (%)
Beginner 8x8x10
Beginner 9x9x10
Intermediate 16x16x40
Expert 30x16x99
Difficulty


first_click_always_zero = 1 | can_flag = 1 | use_mine_count = 0
Any guesses
0 guesses
Sample Size
4x4
5x5
6x6
7x7
8x8
9x9
10x10
Whole grid
Win Rate (%)
Beginner 8x8x10
Beginner 9x9x10
Intermediate 16x16x40
Expert 30x16x99
Difficulty


first_click_always_zero = 1 | can_flag = 1 | use_mine_count = 1
Any guesses
0 guesses
Sample Size
4x4
5x5
6x6
7x7
8x8
9x9
10x10
Whole grid
Win Rate (%)
Difficulty
Beginner 8x8x10
Beginner 9x9x10
Intermediate 16x16x40
Expert 30x16x99
84.5% 85.3% 85.6% 86.3%
67.1% 69.5% 70.0% 71.5%
86.8%
72.3%
95.1% 95.6% 95.7% 95.8% 95.9%
86.7% 88.7% 88.9% 89.1% 89.4%
96.1%
89.8%
81.8% 83.2% 83.7% 83.9% 84.0% 84.1% 84.1% 84.7%
63.4% 67.8% 68.7% 69.2% 69.3% 69.5% 69.6% 70.1%
29.8% 31.8% 32.5% 32.8% 32.9% 33.1% 33.3% 34.8%
9.9% 12.4% 13.2% 13.5% 13.6% 13.9% 14.1% 14.7%

CODE REPOSITORY AND DATA

All of the code for the Minesweeper implementation and solvers, along with the experiment data can be found on the GitHub repository:

`https://github.com/KostasKv/Minesweeper-Learner`

ACCESSING THE DATA The database and all of its data will be accessible as a file database available on the GitHub repository which will have further instructions on how to use it.

To the markers: the database used throughout the project was hosted on a database server which will not allow you to connect to it; the server will ignore any queries from unknown clients. A partial solution to this, which was also intended as a way to overcome the barrier of slow queries, was to save the results of the main queries into a local CSV file and then load the data in from there for whatever jupyter notebooks make use of it.

There is only one notebook that queried the database and it is located at Minesweeper-Learner/data/Main experiment/Main experiment results analysis.ipynb
You can still run the formally submitted version of that notebook but you will have to run the code blocks that load in the data from the local CSV files, and not the blocks preceding which try to query the database.

After the project deadline, the database will be converted into a file database and placed on the GitHub repository along with instructions on how to use it. If you wish to access that data then I would recommend checking that out there.

The aforementioned notebook will be amended to connect to this new server-less database, but beyond that the rest of the repository will remain untouched post-deadline.